The GUMBO Contract Language

Version 1.0
September 2022

Galois, Inc.
111 3rd Avenue South, Suite 100
Minneapolis, MN 55401

Revision History

Revision	Date	Summary
1.0	30 Sept 2022	Initial release of GCL

Contents

1 Introduction

Model-based systems engineering approaches can support the early adoption of mathematical representations of the system under development. The system level requirements may be specified therein and various forms of formal reasoning can provide assurances that the design represented by the model supports the system requirements as specified.

One possible gap between the model and the deployed system is how to be sure that the results of the behavioral analysis applied to the model representation apply to the deployed system.

The High Assurance Modeling and Rapid engineering framework (HAMR) toolset was developed on the Defense Advanced Research Projects Agency (DARPA) Cyber Assured System Engineering (CASE) program to generate the deployable code directly from the architectural model, thereby preserving some forms of traceability between the analytical model results and the deployed system [15]. On the Grand Unified Modeling of Behavioral Operators (GUMBO) Army Phase II SBIR, we extended the HAMR toolchain to bring model level behavioral contracts into code level verification. This extension effort included a behavioral contract language, called GUMBO Contract Language (GCL), that is specified at the model-level. When HAMR code generation is performed, HAMR automatically inserts these contracts into deployable Slang where formal verification by Logika can be performed.

In this document we describe the model-level GCL, the translation of the model to code via HAMR, and the code-level verification in Logika. Before we begin, we provide some background information to help readers new to these concepts.

For installation instructions and use within OSATE, go to `https://github.com/sireum/aadl-gumbo-update-site/blob/master/readme.md`. For support with SysML models, see `https://www.adventiumlabs.com/camet-tools`.

2 Background

As system and software complexity increases, software integration risk has become a limiting factor in the development of complex cyber-physical systems. In a study performed by the System Architecture Virtual Integration (SAVI) initiative, they determined that 70% of errors are introduced at the design phase; most are not found and fixed until integration and test [22]. They reported that fixing those errors late in the development process costs orders of magnitude more than if they had been fixed earlier. We provide an approach to reduce the errors that make it to those late stages by incorporating behavioral specifications derived from system requirements into a tightly integrated modeling and programming paradigm. This shifts development earlier into the design cycle, and thereby eliminates issues earlier when they are less expensive to address. Our approach also enforces and verifies component connectibility and compatibility. This enables multiple development teams to work independently on subsystems and verify that the integration of these components performs as expected.

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 Application Programming Interfaces (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 Architecture Analysis and Design Language (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 GCL, we focus on Slang. GCL contracts are translated to Slang contracts, and Slang implementations of thread component application logic are verified to conform to the contracts.

Figure 1 illustrates the workflow from model to platform. As the components and subsystems are developed in the model, developers can express requirements for each component in GCL. The engineer uses HAMR with our GUMBO extension to autogenerate the infrastructure code and application code method stubs. HAMR weaves the contracts derived from the requirements into that code. At the Slang level, the software engineer will develop the application code in the Integrated Verification Environment (IVE) and receive immediate verification feedback as to its ability to

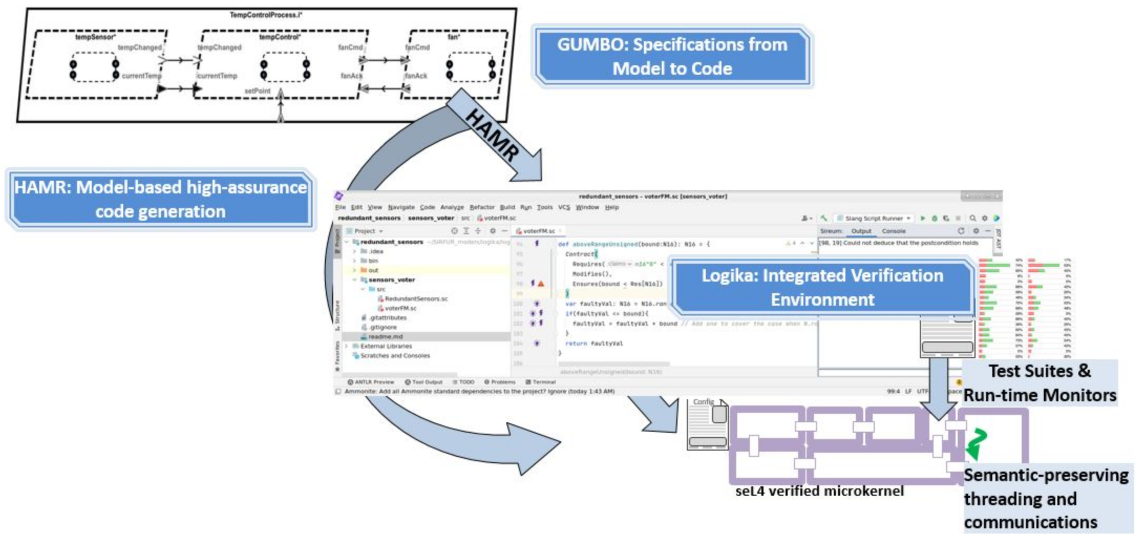

TempControlProcess.i
GUMBO: Specifications from Model to Code
HAMR
HAMR: Model-based high-assurance code generation
Logika: Integrated Verification Environment
Test Suites & Run-time Monitors
seL4 verified microkernel
Semantic-preserving threading and communications

Figure 1: Overview of the model-based specification, auto-generation, and verification workflow.

support those contracts. This verifies that the code deployed on the system adheres to the requirements as specified. The Slang code may then be transpiled to C using a certified compiler (e.g., [17]) and deployed on the desired platform.

At the front-end of the translation pipeline, we have extended the HAMR AADL Intermediate Representation (AIR) to include representations of GCL contracts. This JSON-based representation enables us to support 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 Open Source AADL Tool Environment (OSATE) plug-in.

A detailed overview of HAMR code generation and how code for component application logic is integrated with HAMR's AADL run-time libraries is given in [15] (focusing on Slang code generation) and [11] (focusing on C code generation). A conference submission on the GCL is also available [].

2.1 AADL

The workflow described in this document starts with modeling using the SAE International standard AS5506C [12]. AS5506C 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. AADL provides a precise, tool-independent, and standardized modeling vocabulary of common embedded software and hardware elements using a component-based approach [14]. The AADL standard also describes RTS, a collection of run-time libraries that provide key aspects of threading and communication behavior. As part of GUMBO, we formalized a major subset of the RTS and developed a Slang-based reference implementation [16]. We built GCL and the HAMR extensions with these definitions in mind.

2.2 HAMR

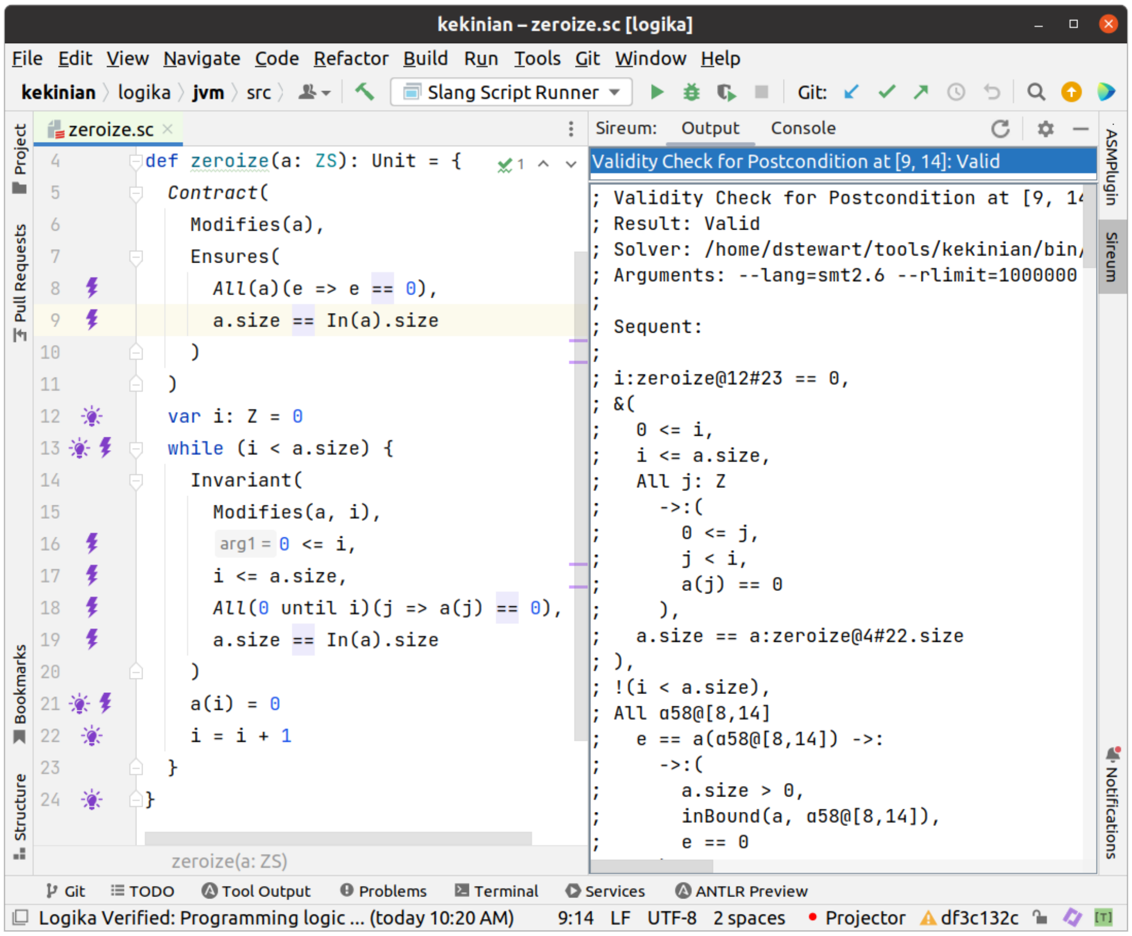
The HAMR tool-kit generates high assurance software from standards-based architectural models for embedded cyber-physical systems [15]. HAMR encodes the system-level execution semantics as specified in standardized models with clear and unambiguous specifications, into infrastructure code suitable for the given target platform. These execution semantics include component interfaces, threading semantics, inter-component communication semantics, scheduling, and more. HAMR implements a C-based back-end for the translation architecture targeting Linux OS and for the seL4 micro-kernel [8]. It also targets Slang, a safety/security-critical subset of Scala [20].

Using model-based information specified in the associated thread type, HAMR generates platform-independent infrastructure, thread code skeletons, and port APIs specific for the thread. The developer will code the thread's application logic in the target programming language. The generated thread-specific APIs serve two purposes: (1) the APIs limit the kinds of communications that the thread can make, thus help ensuring compliance with respect to the architecture, and (2) the APIs hide the implementation details of how the communications are realized by the underlying platform. HAMR documentation, tutorials, and papers are available online [6]. A Kansas State University (KSU) course that uses HAMR extensively is also available [5].

2.3 Logika

Sireum Logika is a highly automated program verifier and a manual proof checker for propositional, predicate, and programming logics [7]. The Logika programming language, Slang, is a subset of Scala that is designed for verification [20]. Logika also provides an automatic program translation to source-traceable and structurally close-to-source C code. This aids in providing a high-assurance toolchain for program correctness from the model, to the program, and further down to machine code.

Logika performs verification compositionally. It accesses multiple back-end model checkers in parallel, including CVC4 [10], CVC5 [9], Z3 [18], and Alt-Ergo [13]. Logika supports incremental and parallel algorithms for compiling, analyzing, and verifying HAMR-generated systems. Proof artifacts, results, and symbolic execution traces are accessible to the developer within the IVE. A view of the IVE is shown in Figure 2.



kekinian – zeroize.sc [logika]
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
kekinian 〉 logika 〉 jvm 〉 src 〉
Slang Script Runner
Git:
zeroize.sc
Sireum: Output Console
Project
Pull Requests
Bookmarks
Structure
ASMPlugin
Sireum
Notifications
zeroize(a: ZS)
Git TODO Tool Output Problems Terminal Services ANTLR Preview
Logika Verified: Programming logic ... (today 10:20 AM) 9:14 LF UTF-8 2 spaces Projector df3c132c

```
4   def zeroize(a: ZS): Unit = {
5     Contract(
6       Modifies(a),
7       Ensures(
8         All(a)(e => e == 0),
9         a.size == In(a).size
10      )
11    )
12    var i: Z = 0
13    while (i < a.size) {
14      Invariant(
15        Modifies(a, i),
16        arg1 = 0 <= i,
17        i <= a.size,
18        All(0 until i)(j => a(j) == 0),
19        a.size == In(a).size
20      )
21      a(i) = 0
22      i = i + 1
23    }
24  }
```

```
Validity Check for Postcondition at [9, 14]: Valid
; Validity Check for Postcondition at [9, 14
; Result: Valid
; Solver: /home/dstewart/tools/kekinian/bin/
; Arguments: --lang=smt2.6 --rlimit=1000000
;
; Sequent:
;
; i:zeroize@12#23 == 0,
; &(
;   0 <= i,
;   i <= a.size,
;   All j: Z
;     ->:(
;       0 <= j,
;       j < i,
;       a(j) == 0
;     ),
;   a.size == a:zeroize@4#22.size
; ),
; !(i < a.size),
; All a58@[8,14]
;   e == a(a58@[8,14]) ->:
;     ->:(
;       a.size > 0,
;       inBound(a, a58@[8,14]),
;       e == 0
```

Figure 2: An example Slang zeroize method shown on the left in the Logika IVE.

The contracts for the method include an *Ensures* clause (pre-conditions), a *Modifies* clause (frame conditions), and a *Requires* clause (post-conditions). For the zeroize method, the developer explicitly specifies that the parameter a, an integer sequence, will be modified. The *Requires* clause states that all elements of the sequence *a* will be zero and the sequence size will not change after the method executes. There are no pre-conditions to this method, so the *Requires* clause is unnecessary. Loop invariants support the compositional proof of the *Requires* statements. The while loop invariants are verified at the beginning and end of the while loop and after each iteration.

By clicking on a lightning bolt (shown on the far left of the figure) or light bulb icon, the developer can view verification conditions, symbolic execution traces, and SMT interactions. This is shown on the right of Figure 2. Upon successful verification, the developer will see a *Logika verified* notification within the IVE.

In the following sections, we describe how a developer can specify system- and component-level requirements in the AADL model using GCL. We also describe how these contracts appear in the HAMR generated infrastructure code.

3 Temperature Control System: Running Example

The following example AADL Temperature Control model is based on the description provided in previous work [15, 16]. The model is available both in the Curated Access to Model-based Engineering Tools (CAMET) release and on GitHub [3].

Figure 3 shows the top level AADL model architecture for a simple temperature controller. The controller maintains a temperature between specified high and low target temperature bounds. A periodic `tempSensor` thread measures the current temperature and transmits the reading on its `currentTemp` data port. When the sporadic (event-driven) `tempControl` thread receives a `tempChanged` event, it will read the value on its `currentTemp` data port and compare it with the most recent set points. For either dispatch protocol, if the current temperature exceeds the high set point, it will send a `FanCmd.On` command to cool the temperature. Similar logic will result in `FanCmd.Off` being sent if the current temperature is below the low set point. In either case, the `fan` acknowledges whether it was able to fulfill the command by sending `FanAck.Ok` or `FanAck.Error` on its `fanAck` event data port.

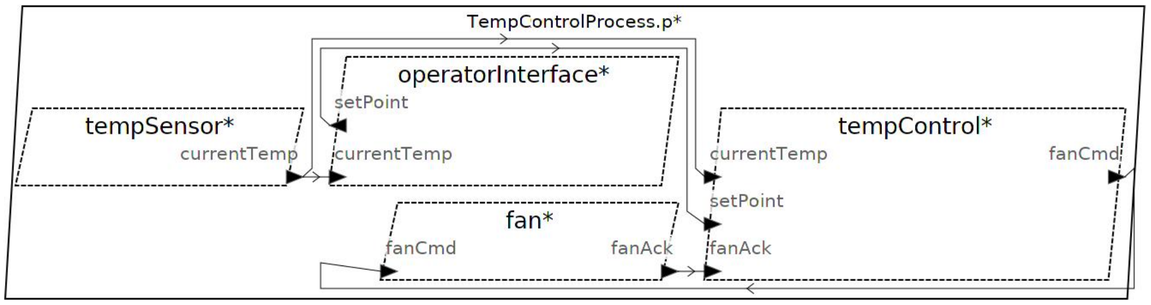


Figure 3: Temperature Control Example AADL Graphical View

AADL provides a textual view to accompany the graphical view. AADL editors such as the Eclipse-based Open Source AADL Tool Environment (OSATE) synchronize the two. The listing below illustrates the *component type* declaration for the above `TempControl` thread. The textual view illustrates that data and event data ports can have types for the data transmitted on the ports. In addition, properties such as `Dispatch_Protocol` and `Period` configure the tasking semantics of the thread.

Lines 1-14 of the listing declares the implementation `TempControl.i` of the `TempControl` component type. When using HAMR, AADL thread component implementations such as `TempControl.i` must have no information in their bodies, which

```
thread TempControl
features
 currentTemp: in data port TempSensor::Temperature.i;
 tempChanged: in event port;
 fanAck: in event data port CoolingFan::FanAck;
 setPoint: in event data port SetPoint.i;
 fanCmd: out event data port CoolingFan::FanCmd;
properties
 Dispatch_Protocol => Sporadic;
 Period => .5 sec;
end TempControl;

thread implementation TempControl.i
end TempControl.i;
```

Figure 4: The sporadic `TempControl` thread component with implementation.

means that there is no further architecture model information for the component. The thread is a leaf node in the architecture model, and further details about the thread's implementation will be found in the source code, not the model.

The listing below illustrates how architectural hierarchy is realized as an integration of subcomponents. The body of `TempControlProcess` type has no declared features because the component does not interact with its context in this simplified example. However, the body of the implementation has subcomponents, and the thread subcomponents are "integrated" by declaring connections between their ports.

```
  process TempControlProcess
    features
      none;
  end TempControlProcess;

  process implementation TempControlProcess.i
    subcomponents
      tempSensor : thread TempSensor::TempSensor.i;
      fan : thread CoolingFan::Fan.i;
      tempControl: thread TempControl.i;
      operatorInterface: thread OperatorInterface.i;
    connections
      ctTStoTC: port tempSensor.currentTemp -> tempControl.currentTemp;
      ctTStoOI: port tempSensor.currentTemp -> operatorInterface.currentTemp;
      tcTStoTC: port tempSensor.tempChanged -> tempControl.tempChanged;
      tcTStoOI: port tempSensor.tempChanged -> operatorInterface.tempChanged;
      fcTCtoF: port tempControl.fanCmd -> fan.fanCmd;
      faFtoTC: port fan.fanAck -> tempControl.fanAck;
      spOItoTC: port operatorInterface.setPoint -> tempControl.setPoint;
  end TempControlProcess.i;
```

AADL editors check for type compatibility between connected ports. HAMR supports data types declared using AADL's standardized Data Model Annex [12]. For example, the data type declarations associated with the temperature data structure are illustrated below.

```
 1 data Temperature
 2   properties
 3   Data_Model::Data_Representation => Struct;
 4 end Temperature;
 5
 6 data implementation Temperature.i
 7   subcomponents
 8     degrees: data Base_Types::Float_32;
 9     unit: data TempUnit;
10
11 data TempUnit
12   properties
13     Data_Model::Data_Representation => Enum;
14     Data_Model::Enumerators=>("Fahrenheit","Celsius","Kelvin");
15 end TempUnit;
```

A standard property indicates that the `Temperature` type is defined as a struct (line 3) and the struct fields and associated types are listed in the data implementation (line 6). The `degrees` field (line 8) has a type drawn from AADL's standardized base type library (32-bit float). The `unit` field (line 9) has an application-defined enumerated type.

AADL provides many standard properties, and allows definition of new properties. Examples of standard properties include thread properties (e.g., dispatch protocols such as periodic, aperiodic, sporadic, etc., and various properties regarding scheduling), communication properties (queuing policies on particular ports, communication latencies between components, rates on periodic communication, etc.), memory properties (sizes of queues and shared memory, latencies on memory access, etc.). User-specified property sets enable developers to define labels for implementation choices available on underlying platforms (choice of middleware realization of communication channels, configuration of middleware policies, etc.). In summary, the property mechanism allows key attributes of the underlying implementation to be exposed to enable model-level analysis and model-level selection of implementation options.

3.1 Requirements Used in Later Examples

TempControl:
The assumptions on the input to the `TempControl` thread are:

- TC-Assume-01: The current temperature range shall be greater than or equal to -70.0 degrees Fahrenheit and less than or equal to 180.0 degrees Fahrenheit.

The requirements that specify the behavior of this thread are:

- TC-Req-01: If the current temperature is less than the set point, then the fan state shall be Off.
- TC-Req-02: If the current temperature is greater than the set point, then the fan state shall be On.
- TC-Req-03: If the current temperature is greater than or equal to the current low set point and less than or equal to the current high set point, then the current fan state is maintained.

Fan:

- F-Req-01: The initial state of the fan shall be Off.

Sensor:

- S-Req-01: The initial value of the sensor upon startup shall be 72.0 degrees Fahrenheit.
- S-Req-02: The operational range of the sensor shall be from -50.0 degrees Fahrenheit to 150.0 degrees Fahrenheit inclusive.

Operator Interface:

- OI-Req-01: The default range of the set point is 70.0 degrees Fahrenheit to 80.0 degrees Fahrenheit.

4 Overview of GUMBO Contract Language

The following is a high-level overview of the GCL to help understand the language in context. The GCL statements are expressed in first order logic. Subsequent sections will provide additional detail on the GCL. Figure 5 illustrates how these contracts fit into a model view of a component.

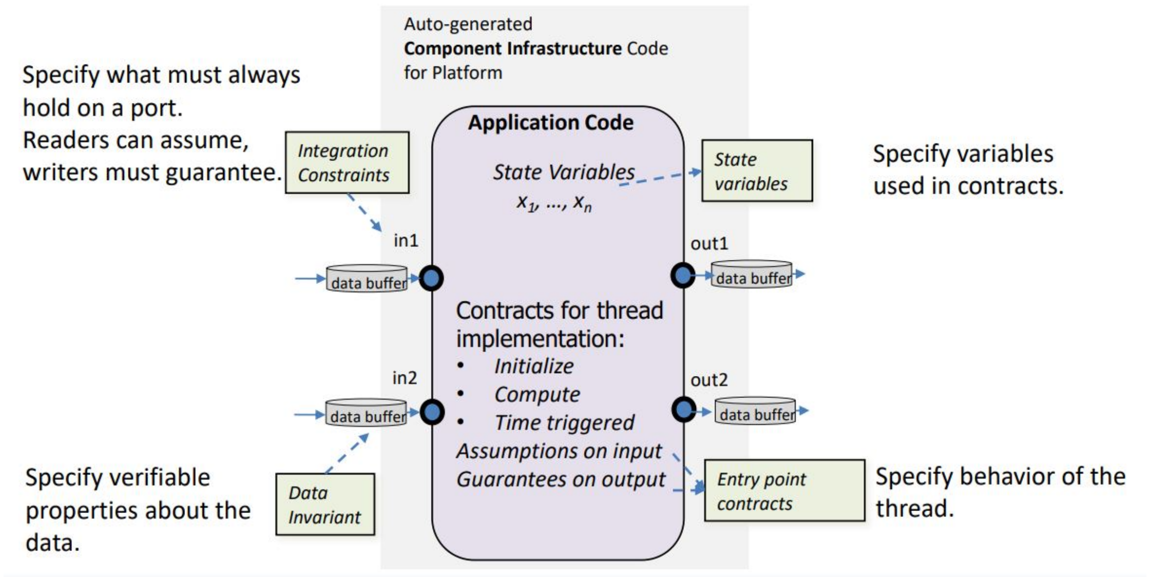


Figure 5: Contracts in the context of auto-generated code.

There are multiple contract categories. They are specified based on the component, data, or port under analysis.

State Variables: A state variable is declared and used within a GCL annex and can be referenced within any contract. It is local to the thread in which it is declared and is translated into a *specification* variable (or *spec* in short). These variables are specific to the verification and are not compiled with the rest of the source code. This is shown in the application code box in the center of Figure 5. More details are given in Section 5.

Data Invariants: specify verifiable and unchanging (hence invariant) properties about data. For example, a value that represents a Kelvin temperature is never negative. This is illustrated in the bottom left of Figure 5. One can also specify constraints over more than one field of a record or struct type. These contracts are defined on a data subcomponent within the model and are verified each time a variable with this particular data type is assigned a value. More details are provided in Chapter 6.

Entry Point Contracts: Each thread component in an AADL model corresponds to application code method stubs in the HAMR generated code. The thread application code is organized into entry points. These are subprograms that are executed when a thread is dispatched. For instance, a periodic thread has an initialize method that is executed when the thread is first initialized. An intialization method is allowed to initialize local variables and write an initial value to output ports. Another entry point for a periodic thread is called *time-triggered*. Once initialization is complete, threads enter the await dispatch state. Initial values of in and out ports are available at first dispatch. Therefore, in the initialization phase, input ports are unavailable and cannot be read. In either of these cases, the developer may specify verifiable behaviors within the contract structures that are specific to each entry point. The structure of the entry point contracts in the GCL is specific to the dispatch protocol. Both sporadic and periodic dispatch protocols are supported in the GCL. This is indicated in Figure 5 in the center of the application code box. More details are given in Section 7.

Integration Constraints: Integration constraints are invariants on the values flowing across ports. This is illustrated in the top left of Figure 5. Integration constraints specify the requirements and assumptions that refer to a single port. These contracts limit to what other ports the component can correctly connect when it is integrated into a system context. It can be helpful during the integration phase of a multi-vendor development effort to determine if component connections are compatible. An input port integration constraint specifies the components assumptions about what it expects to read on the port. A component must guarantee integration constraints on its output ports. More information is given in Section 8. (Note: integration constraints are not fully supported at this time.)

4.1 The Syntax and Semantics of Important Language Constructs

This section provides an overview of the GCL building blocks and simple examples for illustration.

4.1.1 Comments, Identifiers, Strings, and Literals

Comments always start with two adjacent hyphens and span to the end of the line.

```
annex gumbo {**
```

```
    -- Here is a comment

    -- A long comment may be split
    -- onto two or more consecutive lines.
  **};
```

An identifier is defined as a letter followed by zero or more letters, digits, or single underscores (note: an underscore cannot be at the end of an identifier). Some example identifiers are: `stateVar1`, `page_count`, `flag`.

String elements are a sequence of characters within quotation marks. "This is a string."

The following show example boolean and numeric literals:

```
Integer: ..., -101, 0, 15, ...
Real: ..., -8.15, 0.0, 31.977, ...
Floating point: ..., -8.15f, 0.0f, 31.977f, ...
Boolean: T, F
```

4.1.2 Operators and Expressions

We provide the syntax of some commonly used expressions, symbols, and operators. The full grammar is available on GitHub [4]. The arithemtic operators are in Table 1. The logical operators are in Table 2.

Operation	Symbol	Example
Addition	+	`currentTemp.degrees + EPSILON`
Subtraction	−	`currentTemp.degrees - EPSILON`
Multiplication	*	`temp*2`
Division	/	`accumVal/2.0f`
Modulus	%	`arr(i)%2`
Assignment	=	`out = temp*2`
Less than	<	`currentTemp.degrees < MAX_TEMP`
Greater than	>	`currentTemp.degrees > MAX_TEMP`
Less than or equal to	<=	`currentTemp.degrees <= MAX_TEMP`
Greater than or equal to	>=	`currentTemp.degrees >= MAX_TEMP`

Table 1: Arithmetic operators.

Some of the first order logical operators contain short circuit operations. Short-circuiting is when the compiler skips the execution or evaluation of some sub-expressions in a logical expression. The compiler stops evaluating the further sub-expressions as soon as the value of the expression is determined. In this case, the SMT2 solver returns before the entire expression is evaluated during verification.

Operation	Symbol	Example
Negation	!	`!(currentTemp.degrees > 0.0f`
And	&	`flag & (temp > 150)`
Short Circuit And	&&	`flag && (temp > 150)`
Or	|	`flag | (temp > 150)`
Short Circuit Or	||	`flag || (temp > 150)`
Universal	∀, \\all	`\\all i => s(i) > 0`
Existential	∃, \\some	`\\some i => (arr(i)%2 == 0)`
Equivalence	==	`out == 2`
Implication	~>	`flag ~> currentTemp.degrees < 0.0`
Short Circuit Implication	~~>	`flag ~~> currentTemp.degrees < 0.0`

Table 2: Logical operators.

4.1.3 Control Structures

A commonly used control structure in the GCL is the if-else conditional statement:

```
if(expr1) expr2
else expr3
```

The if-else statement may be nested as shown in the function example provided above.

4.1.4 Assume and Guarantee Statements

An *assume* clause states assumptions on the input state and will appear either as preconditions on entry point methods or as an antecedent of a relationship between inputs and outputs. Assume clauses can only reference input ports and declared local state variables. The syntax of an assume statement is shown below.

```
  assume identifier "Optional description string":
      expression;
```

Guarantee clauses state constraints on the output state, which may be conditioned on input state. Guarantees can reference input ports, output ports, and local state variables. This allows for relationships to be specified between inputs and outputs. Guarantee clauses in the GCL are translated to method postconditions. The syntax of a guarantee clause is shown below.

```
    guarantee identifier "Optional description string":
        expression;
```

There are restrictions on what assumption and guarantee expressions may reference depending on what contract they are used in. We will describe these in detail throughout the following sections.

4.1.5 Case Statements

Another type of control structure available for a developer is a `case` statement. A case statement may describe a relationship between inputs and outputs. Depending on the input values, the output changes. There are two ways of describing this relationship. The first is using *cases*, the second is with implications. A case clause contains an assumption over the inputs and a guarantee on the outputs. The syntactic structure of a case statement is shown below.

```
cases
    case identifier1 "Optional descriptive string 1.":
        assume expr;
        guarantee expr;
    case identifier2 "Optional descriptive string 2.":
        assume expr;
        guarantee expr;
```

Case clauses are evaluated in order. For a particular case, if the assumes clause is true for the method pre-state, then the guarantees clause must be true in the post-state. Case clauses need not be mutually disjoint, in other words, multiple case clauses may be true simultaneously. In this situation, all of their respective guarantee clauses are enforced. The case assume clauses are not required to be exhaustive. For example, There may be a pre-state that does not satisfy any case assumes clause.

4.1.6 Functions

Developers may define functions corresponding to Slang `@strictpure` and `@pure` methods [20]. Strict pure and pure methods may simplify complex specifications. Strictpure methods can be directly translated to logical representations, and therefore they do not require explicit functional behavior contract specifications for compositional verification purposes. Thus, they can be treated directly as specifications. The GCL supports only a subset of what Slang `@strictpure` methods support. A method body may only contain a single expression or an if-else statement returning a value. The syntax of a function definition is as follows.

```
def functionName(param1: paramType, ..., paramN: paramType): returnType := Expr;
```

Functions may be defined locally within a thread component annex or globally in a specification file. Globally scoped functions may be used within any thread component annex, as long as the correct package is included. We show two functions one may define for the temperature control system.

```
  def inRange(temp: TempSensor::Temperature.i) : Base_Types::Boolean := (MIN_TEMP <
      temp) & (temp < MAX_TEMP);

  def toFahrenheit(temp: TempSensor::Temperature.i): Base_Types::Real :=
       (temp - 32.0)*5.0/9.0;
```

Alternatively, one may define a `@pure` method by appending a contract. A contract specifies the *assume* clauses (pre-conditions), and *guarantee* clauses (post-conditions). These assumptions and guarantees are translated to `Requires` and `Ensures` clauses in Logika. An example of a method defined in the GCL with contract annotation is as follows.

```
def toFahrenheit(temp: TempSensor::Temperature.i): TempSensor::Temperature.i :=
  Contract(
    guarantees(
      isFahrenheit(Res)
  )
  if (temp.unit == TempSensor::Unit.Kelvin)
    TempSensor::Temperature.i(f32"1.8" * (temp.degrees - f32"273") + f32"32",
      TempSensor::Unit.Fahrenheit)
  else
    if (temp.unit == TempSensor::Unit.Celsius)
    TempSensor::Temperature.i((temp.degrees * f32"1.8") + f32"32",
          TempSensor::Unit.Fahrenheit)
    else
      temp;
```

4.1.7 Specification Keywords

In the GCL, there are separate specification (spec) sections that can be defined using the keywords in italics below. These sections subdivide the annex.

- A *state* variable declaration defines a state variable that can be referred to within the annex.
- A data *invariant* enforces some constraint on a data subcomponent. All constructed and updated values of this data subcomponent must adhere to this constraint.

- An *initialize* section defines the inital default values on the output ports.
- A *compute* entry point contract states relationships between input port values and output port values. In short, these specify the functional behavior constraints on values flowing in and out of a component during the compute entry point.
- A *function* section contains functions that may be used within any contract of an annex. Functions can be defined locally within an annex or globally.
- A port *integration* constraint indicates a property that should always hold for data items flowing in and out of a port.

The behavior of each component can be defined by using these sections. Each spec section is described in detail in later sections of this document.

4.1.8 Input State

To distinguish the input state of local state variables from its output state when a state variable is referenced in a guarantee clause, the `In(...)` construct is used to reference the input state of a state variable. When `In(...)` is not used, the output state of the variable is referenced. The syntax of an `In` statement is shown below.

```
   In(stateVar)
```

The `In(...)` designator is not used in assume clauses. The assume clause can only reference aspects of the input state. There is no need to use `In(...)` for port references. In the subset of AADL supported by HAMR, each port must be declared as either an `in` port or an `out` port.

4.1.9 Modifies Clause

For a contract, the `modifies` clause specifies the declared local state variables that may be modified within the respective contract. These are commonly known as *frame conditions* because they “frame” which part of the thread’s variable state may be modified and which will not. Any logical constraints on variables that are not listed in the modifies clause that hold in the input state will continue to hold in the output state. For any variable listed in the modifies clause, the only constraints that hold in the output state are those explicitly stated in entry point guarantee clauses. An example of the syntax of a modifies clause is shown below.

```
 modifies stateVar1, stateVar2;
```

4.1.10 Must, No, and May Send Statements

A statement specific to a sporadic thread compute entry point is a `MaySend(...)`, `NoSend(...)`, or `MustSend(...)` statement. (Note: `MaySend(...)` statements are currently unsupported in the GCL.) These statements specify events that may, must, or will not be sent on an event output port if certain conditions apply. The argument to a may or must send statement will always be the name of an event output port. An example of the syntax of these statements is shown below.

```
MaySend(eventOutPortName (, varToSend)?)
NoSend(eventOutPortName)
MustSend(eventOutPortName (, varToSend)?)
```

For may and must send statements, one may optionally add a state variable or port value to the parameter list to specify the value that must (or will not) be sent on the event data port.

5 State Variables

State variables are used to declare local variables to the thread annex. They can be referenced by any clause within that thread's annex. The syntax of a state variable section in the annex is as follows:

```
annex gumbo{**
  state:
    stateVarId: varType;
    stateVarId2: varType;
**};
```

In some situations, a state variable may be used to save the state of event data ports. Upon dispatch a thread will read all input data ports; therefore, those values are available and fresh at each dispatch. Event data ports are only read upon the arrival of a new event. There are times when a thread will require access to the latest event data port value, but will not read that port within a given dispatch. A state variable can be updated at the time of event arrival and used in later dispatches to refer to the last event data port value.

Given that a periodic component will read input data ports at each dispatch, the only local state variable that is necessary to specify the requirements is the `latestFanCmd`. This is necessary to maintain due to requirement *TC-Req-03: if the temperature remains within low and high set points, then the fan state should remain the same as what it was in the last dispatch*. We need to save the setting for the fan command in the current dispatch so that we can refer to it in the next dispatch. To this end, we can create a state variable to hold this value. The remaining requirements reference values that are read from the input ports.

```
state:
    latestFanCmd: CoolingFan::FanCmd;
```

By explicitly specifying that the component will declare and maintain the value of a local variable `latestFanCmd`, we can reference its pre-state in an assumes clause and both its post-state (using *In(...)*) and pre-state in a guarantees clause.

5.1 Model-level Syntax and Examples

The sporadic temp control thread (shown in Figure 4) has an input data port and three input event data ports. State variables are defined for each of the event data ports.

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

As we show in Section 7.4, each of these state variables are updated upon the arrival of their respective event. They may then be referenced on any dispatch and will hold the latest event data port value.

5.2 Code-level Verification

The state variables defined for a thread component are translated as object variables within the thread application code. The thread application code method stubs are found in the `src/main/component/` directory and are ordered by thread component name. The state variables for the sporadic `TempControl` thread are given in the listing below.

```
object TempControl_s_tcproc_tempControl {

  // BEGIN STATE VARS
  var currentSetPoint: TempControlSoftwareSystem.SetPoint_i =
      TempControlSoftwareSystem.SetPoint_i.example()

  var currentFanState: CoolingFan.FanCmd.Type = CoolingFan.FanCmd.byOrdinal(0).get

  var latestTemp: TempSensor.Temperature_i = TempSensor.Temperature_i.example()
  // END STATE VARS

  def initialise(api: TempControl_s_Initialization_Api): Unit = {...}
  ...
```

Each of the methods within the thread can access these local variables, reassign values, and refer to them within contracts or application code. The `example()` method for each datatype returns an example value of that type. One can access these methods within the HAMR generated code base.

6 Data Invariants

Requirements often specify invariants over data flowing through a system. In order for the data to make sense within the system context, invariants are given. Example uses of data invariants include constraining the range of a numeric type. For instance, a sensor value lies within the range of 0 and 100 degrees Celsius. One can also specify constraints over more than one field of a record or struct type. An example is a record representing dates in Day/Month/Year format with three integer fields. This data has implicit invariants that are required for the date to be reasonable (e.g., the day field must be constrained differently depending on the month field). An invariant can also specify a well-formed condition on a composite data structure. For instance, an array of coordinate values representing waypoints; longitudinal values stay with 180 degrees of zero and latitudinal values within 90 degrees.

Data invariants provide finer-grained constraints on the set of values that belong to a data type D. Data invariant constraints can only reference D values or sub-elements of D (e.g., struct fields or array elements). They can not reference other system state or types not used within D.

6.1 General Concepts

In large-scale development projects, common data type specifications are often organized and documented in data dictionaries to be shared across development teams. Terms such as "data models" (as in Future Airborne Capabilities Environment (FACE) [2] or Department of Defense Architecture Framework (DoDAF) [1] or "data view" or "information view" in view-based modeling frameworks [21] often extend descriptions of data representations with textual descriptions of data constraints. In some cases, constraints languages like UML's Object Constraint Language (OCL) [19] may be used to formalize constraints. We provide support for these types of constraints that goes well beyond current industrial tools. Specifically, we enable formal machine-readable specification of such constraints along with multiple forms of automated verification at the model level and code level.

At the model level, when a developer writes a data invariant in a data component type D, the specification intent is that wherever values of an instance of type D flow through the system, they satisfy the invariants specified for D. Since a model-level verification tool does not have visibility into the system's implementation to enforce the "wherever the value flows" concept, it will typically limit its reasoning to the model's component boundaries, which are explicitly reflected in the model. That is, it may check that whenever values flow from an output port of type D or into an input port of type D, they must satisfy the specified data invariants of D.

For values of instances of D flowing into a component C, other specifications for C may rely on the fact that D data invariants are satisfied. Enforcement of data invariants at the code level may vary based on the target programming language and model checking technology. An important aspect of verification is that data values should satisfy corresponding invariants when being exchanged through APIs for sending and receiving over ports. During verification, Logika will check that whenever values of D are created and initialized, the fully initialized value should satisfy any associated data invariant constraints. Likewise, whenever a value of of an instance of D is updated, the new value satisfies the invariant.

6.2 Model-level Syntax and Examples

The basic syntax of an invariant is shown below.

```
annex GUMBO{**
  invariants:
    inv invariantId1: expr;
    inv invariantId2: expr;
**};
```

Multiple invariants may defined within an annex. Each begin with the keyword `inv` and have their own unique identifier. The expression defines the logical invariant on the data.

We present two examples of data invariant specifications for the `Temperature` and `SetPoint` types of the temperature control example.

```
data implementation Temperature.i
  subcomponents
    degrees: data Base_Types::Float_32;
  annex GUMBO{**
    invariants
      inv absZeroInvariant:
        f32"-459.67" <= degrees;
  **};
end Temperature.i;
```

In the listing above, a single invariant is specified over the `Temperature` data type. The inequality references the `degrees` field and the expression defines the Boolean constraint.

```
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_Low_High_Bounds:
         f3"50.0" <= low.degrees and  high.degrees <= f32"110.0";
      inv "SetPoint_Low_High_rdering.":
         low.degrees <= high.degrees;
  **};
end SetPoint.i;
```

For the `SetPoint` type in the listing above, two invariants are defined. The first defines bounds on the low and high values of the set point. The second ensures that the low set point is never greater than the high set point. The two invariants could also be specified in a single invariant using a conjunction.

The SetPoint type is a struct, and following the AADL Data Model Annex approach, struct fields are specified in data component implementations. The data invariant is likewise specified there. In other cases, the invariant may be defined in the component type. The invariant is defined in the AADL component specification (type or implementation) in which the root data type is defined. In the case of an AADL base data type, the invariants are defined in the component type.

An annex clause in a data component may contain at most one `invariant` section, and this section may contain one or more `inv` clauses.

6.3 Code-Level Verification

Using the HAMR code-generation framework, the data invariants are translated into specification statements in Logika. All data invariants are found within the relevant data object in the directory `src/main/data`. An example of the `Temperature` data type class and Logika spec statement is shown below.

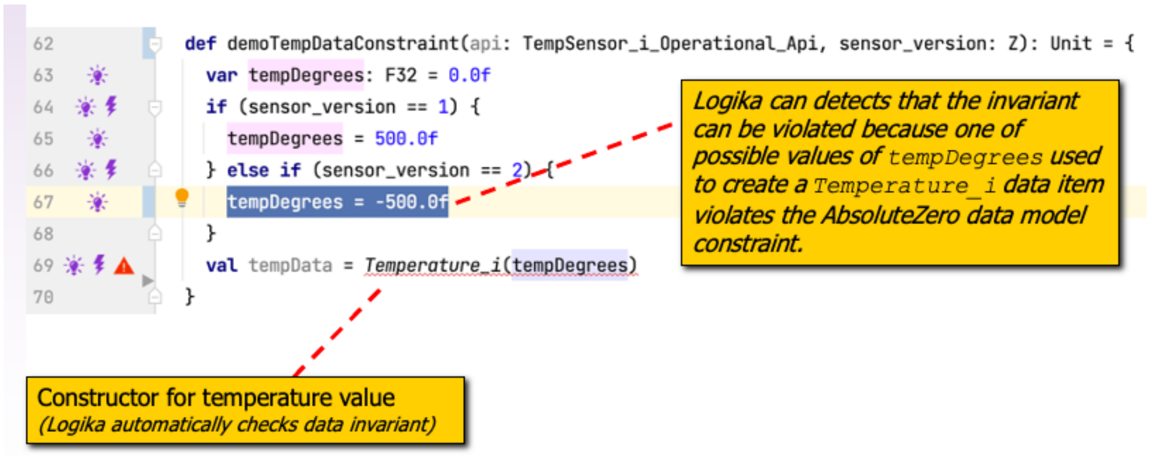
```
@datatype class Temperature_i(
  degrees : F32) {
  @spec def temperature_Inv = Invariant(
    degrees >= f32"-459.67"
  )
}
```

Notice that this specification is translated directly from the model-level specification. Anytime the temperature datatype is constructed, it must be greater than or equal to absolute zero. The `SetPoint` data constraint in Logika is shown in the listing below.

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

For each data type invariant, the Logika verifier checks that each time a value of that type (e.g., `Temperature_i`) is constructed, the constructed value satisfies the invariant. Since Slang data types are immutable, this checking ensures that any `Temperature_i` constructed in the system will satisfy its invariant. Thus, successful Logika verification guarantees that whenever a temperature value flows out an AADL port with type D, the value will satisfy the D GCL data invariant. This is because Logika guarantees that the invariant holds from the time the value is originally constructed.

Figure 6 illustrates Logika checking of data invariants on a simple demo method inserted into a HAMR-generated Slang system.



```
62  def demoTempDataConstraint(api: TempSensor_i_Operational_Api, sensor_version: Z): Unit = {
63    var tempDegrees: F32 = 0.0f
64    if (sensor_version == 1) {
65      tempDegrees = 500.0f
66    } else if (sensor_version == 2) {
67      tempDegrees = -500.0f
68    }
69    val tempData = Temperature_i(tempDegrees)
70  }
```

Logika can detects that the invariant can be violated because one of possible values of `tempDegrees` used to create a `Temperature_i` data item violates the AbsoluteZero data model constraint.

Constructor for temperature value
(Logika automatically checks data invariant)

Figure 6: Logika verification of the data invariant on the `Temperature_i` constructor. Verification failed due to a possible data constraint violation.

Line 69 includes a call to the `Temperature_i` constructor. Conditional logic above assigns several possible values to the `tempDegrees` variable passed to the constructor. Logika is able to detect that assignment at line 67 could give the `tempDegrees` variable a degrees value less than absolute zero which, when used in the constructor at line 69 would cause the `Temperature_i` invariant to be violated.

7 Entry Point Contracts

A thread's application code is organized into entry points. This is the place in the thread's application code where the execution begins. The *initialize entry point* is called during the system's initialization phase and the *compute entry point* is called during the threads normal dispatch (see Figure 7). Other entry points include finalization, handling faults, performing mode changes, etc. The GCL provides support for adding contracts to the initialize and compute entry points.

This section will describe the general concepts of entry points and provide examples of their use in both AADL and the generated Slang with Logika contracts. For examples of language syntax and semantics, see Section 7.2.

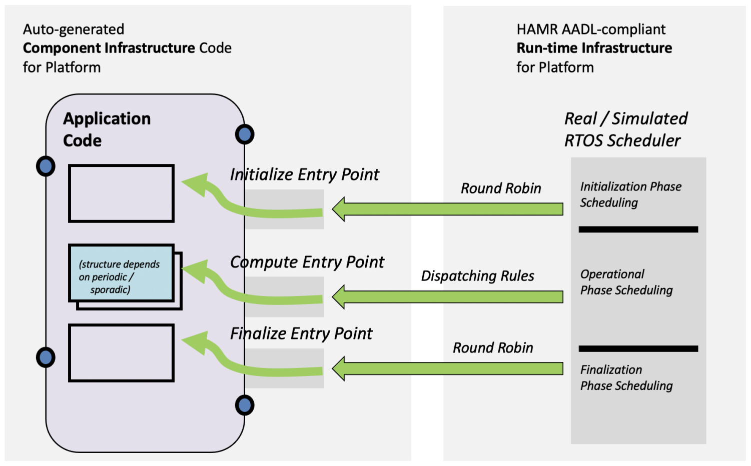


Figure 7: AADL Entry Point Concepts

7.1 General Concepts

Entry points correspond to blocks of application code that may be dispatched by the run-time services to provide the application-level functionality for the component. The structure of the entry point contracts change based on the category of entry point and the dispatch protocol for the entry point. For example, an initialize entry point will not read input ports, but it will initialize local state and output ports. A compute entry point may read input ports and the values of local state variables and update output ports and state variables accordingly. Detailed examples of this will be provided in subsequent sections.

GUMBO entry point contracts enable developers to specify key properties of the functionality of each entry point as illustrated in Figure 8.

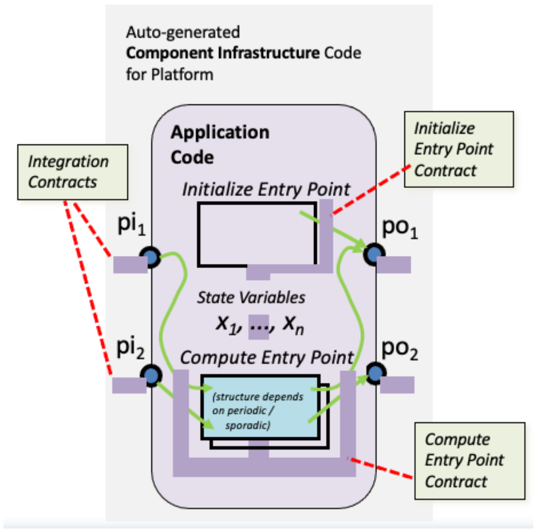


Figure 8: GUMBO Entry Point Contracts Concepts

The application code contains initialize and compute entry points. Each has access to local state variables $(x_1, \ldots, x_n)$. The structure of entry point contracts will differ depending on the entry point category (e.g., initialize, compute) and dispatch protocol (e.g., sporadic, periodic). Entry point contracts impose additional constraints beyond the component's integration contracts. As described in Section 8, the integration constraints apply to each input port read by a thread. This is an invariant on port values that must hold for all entry points. Whereas each integration constraint only applies to a single port, an entry point contract may state relationships between input and output port values and may also constrain local state variables.

Initialize entry point code is not allowed to read input ports or values of local state variables. It can, however, write to output ports and initialize state variables.

Compute entry point application code may specify a contract that relates component input ports to output ports. This is represented in Figure 8 as input ports pi_1, pi_2 and output ports po_1, po_2. In addition, the compute entry point contract may put constraints on the input and output states of component local variables.

The structure of a compute entry point may change depending on the thread's dispatch protocol. Periodic components are triggered by the advent of the start of a components specific period, whereas a sporadic component is triggered by the arrival of an event or one or more input event or event data ports. Their corresponding compute entry points will need to specify behaviors in different ways. We describe these differences in detail in Section 7.2.

7.2 Model-level Syntax and Examples

The temperature control system was described in Section 3. We focus on the `TempControl` thread in this section and show entry point contracts for both the periodic and sporadic versions of the thread. When the sporadic `TempControl` thread receives a `tempChanged` event, it will read the value on its `currentTemp` data port and change the fan command depending on that value and the current `setPoint`.

The requirements for this thread are listed in Section 3.1. We refer to these requirements throughout the section. Given that the `TempControl` component has an output data port read by the fan component, the initialization of the `TempControl` thread must be compatible with the initialization requirements of the `Fan` and `Sensor` threads.

Assumptions and Guarantees Section 4.1.4 has more information regarding assumptions and guarantees. There are restrictions on what assumption and guarantee expressions may refer to within entry points:

- Assumptions can refer to:
  - input ports
  - pre-states of declared local state variables
- Guarantees can refer to:
  - output ports
  - post-states of declared local state variables
  - values of input ports (to relate inputs to outputs)
  - pre-state values of declared local state variables (using *In(...))*

7.3 Entry Point Contracts for Periodic Threads

The `TempControl` thread may be developed using a periodic dispatch protocol. To illustrate the changes in entry point contracts, we will show both periodic and sporadic dispatch protocols and describe necessary model and contract changes to the thread component in each case. We will begin with the periodic dispatch protocol.

A graphical representation of the periodic version of the temperature control system is shown in Figure 9. We note that all threads of the periodic version of the temperature control system are periodic and not mixed. All ports in the periodic variant are data ports.

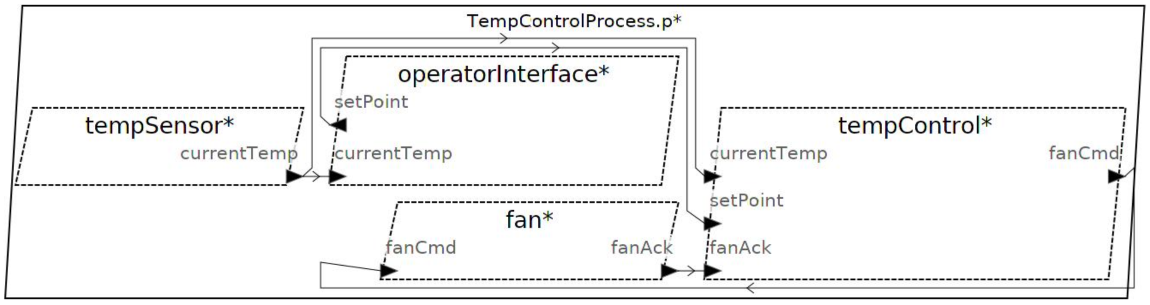


Figure 9: Temperature control system with periodic components – AADL graphical view

The AADL component type definition of the `TempControlPeriodic` thread is shown below for convenience.

```
thread TempControlPeriodic
  features
    -- ==== INPUTS ====
    currentTemp: in data port TempSensor::Temperature.i;
    fanAck: in data port CoolingFan::FanAck;
    setPoint: in data port SetPoint.i;
    -- ==== OUTPUTS ====
    fanCmd: out data port CoolingFan::FanCmd;
  flows
    ct2fc: flow path currentTemp -> fanCmd;
    sp2fc: flow path setPoint -> fanCmd;
    fa2sink: flow sink fanAck;
  properties
    Dispatch_Protocol => Periodic;
    Period => 1 sec;
end TempControl;
```

The `TempControlPeriodic` thread may read three input ports upon dispatch: `currentTemp`, `fanAck`, and `setPoint`. Based on the values of these inputs, the thread will write to the `fanCmd` output port.

7.3.1 State Variables

Given that a periodic component will read input data ports at each dispatch, the only local state variable that is necessary to specify the requirements is the `latestFanCmd`. This is necessary to maintain due to requirement *TC-Req-03: if the temperature* *remains within low and high set points, then the fan state should remain the same as what it was in the last dispatch.* We need to save the setting for the fan command in the current dispatch so that we can refer to it in the next dispatch. To this end, we can create a state variable to hold this value. The remaining requirements reference values that are read from the input ports.

```
state:
    latestFanCmd: CoolingFan::FanCmd;
```

By explicitly specifying that the component will declare and maintain the value of a local variable `latestFanCmd`, we can reference its pre-state in an assumes clause and both its post-state (using *In(...)*) and pre-state in a guarantees clause.

7.3.2 Integration Constraints

As described in Section 8, integration constraints are the invariants on the values flowing across ports that apply to all entry points. When specifying the integration constraints, we identify the requirements and assumptions that refer to a single port. Recall that integration constraints do not relate inputs to outputs, but rather state an invariant over a single port value. For this thread, we have an assumption that the `TempControlPeriodic` thread can make about the values it reads on the `currentTemp` input port. Note that these constraints do not change whether a thread is sporadic or periodic. The constraints are shown below.

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

Recall that an integration constraint has restrictions on what may be referenced within the assume or guarantee expression. Assumptions can only refer to a single input port. Guarantees can only refer to a single output port.

7.3.3 Initialize Entry Points

The initialize entry point will set the initial values of all local state variables and output ports. In this example, `TempControlPeriodic` thread has one state variable and the output port read by the `Fan` component. The initialize entry point contract is shown below.

```
initialize
    modifies lastestFanCmd;
    guarantee initFanCmdStateVar "Initialize state variable.":
       latestFanCmd == FanCmd.Off;

    guarantee initFanCmd "Initial fan command.":
       fanCmd == FanCmd.Off;
```

The `TempControlPeriodic` thread has a single output port that will be read by the `Fan` component. The fan has a requirement stating that its initial state shall be Off (F-Req-01). Thus, the initial value placed on this output data port will conform to the `Fan` requirement.

7.3.4 Compute Entry Point: Cases Syntax

In many cases, a compute entry point clause describes a relationship between inputs and outputs. Depending on the input values, the output changes. To this end, we have defined two ways of describing this relationship. The first is using *cases*. A case clause contains an assumption over the inputs and a guarantee on the outputs. The syntactic structure of a case statement is shown below.

```
cases
    case identifier1 "Optional descriptive string 1.":
        assume expr;
        guarantee expr;
    case identifier2 "Optional descriptive string 2.":
        assume expr;
        guarantee expr;
```

Case clauses are evaluated in order. For a particular case, if the assumes clause is true for the method pre-state, then the guarantees clause must be true in the post-state. Case clauses need not be mutually disjoint, in other words, multiple case clauses may be true simultaneously and in this situation, all of their respective guarantee clauses are enforced. Also the case assume clauses may not be exhaustive. There may be a pre-state that does not satisfy any case assumes clause.

Many requirements for the `TempControl` component can be specified as an implication in which the antecedent references inputs and the conclusion references outputs. The case clauses work well for this kind of requirement. The full compute entry point specification using case statements is shown below.

```
compute
  modifies latestFanCmd;

  cases
    case TC-Req-01 "TC-Req-01: If the current temperature is less than the set point
        ,
           then the fan state shall be Off.":
      assume currentTemp.degrees < setPoint.low.degrees;
      guarantee latestFanCmd == FanCmd.Off & fanCmd == FanCmd.Off;

    case TC-Req-02 "TC-Req-02: If the current temperature is greater than the set
        point,
           then the fan state shall be On.":
```

```
      assume currentTemp.degrees > setPoint.high.degrees;
      guarantee latestFanCmd == FanCmd.On & fanCmd == FanCmd.On;

    case TC-Req-03 "TC-Req-03: If the current temperature is greater than or equal
        to the current
           low set point and less than or equal to the current high set point,
           then the current fan state is maintained.":
      assume currentTemp.degrees >= setPoint.low.degrees
           & currentTemp.degrees <= setPoint.high.degrees;
      guarantee latestFanCmd == In(latestFanCmd) & fanCmd == latestFanCmd;
```

7.3.5 Compute Entry Point: Implication Syntax

Alternatively, one can use guarantees and implications to specify the relationship between input and output. This alternative syntax is shown below.

```
compute
  modifies latestFanCmd;

  guarantee TC-Req-01 "If the current temperature is less than the set point,
           then the fan state shall be Off.":
      (currentTemp.degrees < setPoint.low.degrees)
     ~>: (latestFanCmd == FanCmd.Off & fanCmd == FanCmd.Off);

  guarantee TC-Req-02 "If the current temperature is greater than the set point,
           then the fan state shall be On.":
      (currentTemp.degrees > setPoint.high.degrees)
     ~>: (latestFanCmd == FanCmd.On & fanCmd == FanCmd.On);

  guarantee TC-Req-03 "If the current temperature is greater than or equal to the
      current
      low set point and less than or equal to the current high set point,
      then the current fan state is maintained.":
    (currentTemp.degrees >= setPoint.low.degrees
       & currentTemp.degrees <= setPoint.high.degrees)
   ~>: (latestFanCmd == In(latestFanCmd) & fanCmd == latestFanCmd);
```

7.4 Entry Point Contracts for Sporadic Threads

A sporadic thread's compute entry point is dispatched upon the arrival of an event. The event dispatch corresponds with a specific event handler in the application code of the thread. Some contracts must hold for all handlers and all event dispatches. Other contracts are event specific. The structure of a compute entry point for a sporadic thread corresponds to this concept. The `TempControlSporadic` thread component type in AADL is given in the listing below.

The main differences between the periodic and sporadic components are the category of input and output ports, and obviously the dispatch protocol.

```
 1 thread TempControlSporadic
 2   features
 3     -- ==== INPUTS ====
 4     currentTemp: in data port TempSensor::Temperature.i;
 5     tempChanged: in event port;
 6     fanAck: in event data port CoolingFan::FanAck;
 7     setPoint: in event data port SetPoint.i;
 8     -- ==== OUTPUTS ====
 9     fanCmd: out event data port CoolingFan::FanCmd;
10   flows
11     ct2fc: flow path currentTemp -> fanCmd;
12     tc2fc: flow path tempChanged -> fanCmd;
13     sp2fc: flow path setPoint -> fanCmd;
14     fa2sink: flow sink fanAck;
15   properties
16     Dispatch_Protocol => Sporadic;
17     Period => 1 sec;
18 end TempControlSporadic;
```

Figure 10: The AADL component definition of the sporadic `TempControl` thread.

7.4.1 State Variables

The state variables defined for `TempControlSporadic` are shown in Figure **??**.

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

The `currentSetPoint` local state variable will be set upon the arrival of the event `setPoint` defined on line 7 of Figure 10. This ensures that the last set point value can be accessed upon the triggering of any other event.

As `FanCmd` events are sent on the event output port of the `TempControlSporadic` thread (line 9 of Figure 10), the command is saved to the local state variable `currentFanState` state variable. Any contracts that reference the state of the fan will access that value via `currentFanState` state variable.

The `currentTemp` data port (line 4 of Figure 10) is read when a `tempChanged` event (line 5 of Figure 10) triggers dispatch. If no `tempChanged` event arrives, we can access the last known temperature from the local state variable `latestTemp`.

7.4.2 Integration Constraints

The integration constraints defined for the sporadic component are equivalent to those defined on the periodic component (Section 7.3.2). The integration constraints specify the invariants on the values found on the input ports of the component. See

Section 8 for more details on integration constraints.

7.4.3 Initialize Entry Points

The initialize entry point will set the initial values of all local state variables and output ports. The initialize entry point for the `TempControlSporadic` thread is shown below.

```
initialize
  modifies currentSetPoint, currentFanState, latestTemp;
  guarantee OI-Req-01 "The default range of the set point is -70.0 degrees
            Fahrenheit to 80.0 degrees Fahrenheit.":
    currentSetPoint.low.degrees == f32"-70.0" & currentSetPoint.high.degrees == f32"
        80.0";
  guarantee F-Req-01 "The initial state of the fan upon startup is Off.":
    currentFanState == FanCmd.Off;
  guarantee S-Req-01 "The initial value of the sensor upon startup shall be
            72.0 degrees Fahrenheit.":
    latestTemp == f32"72.0";
```

The requirements of the temperature control system should state initial values for temperature and set point ranges. These are used to set the initial state of the variables. In all cases, the initialization values must be compatible across the system. In this example, we have initialization values defined in the Operator Interface (OI) requirements, the Fan (F) requirements, and the Sensor (S) requirements. These are stated in the guarantee description strings of the listing above.

Note that we do not write an initial value to an output port. The reason is because there are no data output ports defined for the `TempControlSporadic` component. We do not send events in the initialize entry point.

7.4.4 Compute Entry Points

The compute entry points for a sporadic component may consist of two different sets of contracts. One set specifies properties reflecting control laws that should always hold, no matter what event triggers dispatch the thread. Therefore, these contracts are stated as independent guarantee clauses (i.e., not associated with a handler). The second set of contracts are event handler specific. Handler contracts specify the set of requirements that must hold for a particular event handler. We begin by showing the guarantees that should hold regardless of the event that triggers dispatch.

```
compute
  modifies currentSetPoint, currentFanState, latestTemp;
  guarantee TC-Req-01 "If the current temperature is less than the set point,
          then the fan state shall be Off.":
    latestTemp.degrees < currentSetPoint.low.degrees ~>: currentFanState == FanCmd.
        Off
```

```
 6  guarantee TC-Req-02 "If the current temperature is greater than the set point,
 7           then the fan state shall be On.":
 8    latestTemp.degrees > currentSetPoint.high.degrees ~>: currentFanState == FanCmd.
        On
 9  guarantee TC-Req-03 "If the current temperature is greater than or equal to
10           the current low set point and less than or equal to the current
11           high set point, then the current fan state is maintained.":
12    (latestTemp.degrees >= currentSetPoint.low.degrees)
13      and (latestTemp.degrees <= currentSetPoint.high.degrees)
14        ~>: (currentFanState == In(currentFanState))
15  guarantee mustSendFanCmd "If the local record of the fan state was updated,
16           then send a fan command event with this updated value.":
17    In(currentFanState) != currentFanState ~>: MustSend(fanCmd)
```

Notice that the guarantees in this block are very similar to the ones specified in the periodic component, but in this case, refer to state variable values instead of data port inputs. The last guarantee on line 15 introduces a `MustSend` statement. If the local state variable `currentFanState` has been updated, then a corresponding `fanCmd` must be sent on the event output port.

The handler specific contracts can define contracts that will be verified upon dispatch of the thread upon a specific event arrival. A handler for each event input port is not required. While the earlier described guarantees apply for all handlers, the handler contracts allow the compute entry points to be extended with additional constraints that must hold only for a particular handler.

After translation into Slang, each handler's contract is formed by conjoining the compute entry point general clauses with the contract clauses given in the handlers. The handler contracts for each event input port is shown below.

```
handle setPoint:
modifies currentSetPoint;
guarantee setPointChanged:
  currentSetPoint == setPoint;
guarantee lastTempNotModified:
  latestTemp == In(latestTemp);

handle tempChanged:
modifies latestTemp;
guarantee tempChanged:
  latestTemp == currentTemp;
guarantee setPointNotModified:
  currentSetPoint == In(currentSetPoint);

handle fanAck:
modifies currentFanState;
guarantee setPointNotModified:
  currentSetPoint == In(currentSetPoint);
guarantee lastTempNotModified:
  latestTemp == In(latestTemp);
guarantee currentFanState:
  currentFanState == fanCmd;
```

The contract starting on line 1 specifies the behavior of the `setPoint` handler. If this event triggers the dispatch of the `TempControlSporadic` thread, then we change the local state variable accordingly (lines 3-4) and we ensure that the state variable `latestTemp` is maintained.

7.5 Code-Level Verification

Local state variables and all entry point contracts are automatically inserted into the thread component application code in the `src/main/component` directory. For the sporadic `TempControl` thread, the state variables are at the thread object level, and the entry point contracts are inserted into the entry point method. The state variable translation is shown for convenience in the listing below.

```
object TempControl_s_tcproc_tempControl {

  // BEGIN STATE VARS
  var currentSetPoint: TempControlSoftwareSystem.SetPoint_i =
      TempControlSoftwareSystem.SetPoint_i.example()

  var currentFanState: CoolingFan.FanCmd.Type = CoolingFan.FanCmd.byOrdinal(0).get

  var latestTemp: TempSensor.Temperature_i = TempSensor.Temperature_i.example()
  // END STATE VARS

  def initialise(api: TempControl_s_Initialization_Api): Unit = {...}
  ...
```

The systems engineer developed contracts in the GCL for each handler. These can be seen in full in the temperature control example provided with this document. We show a single handler contract after translation to Slang.

```
def handle_setPoint(api: TempControl_s_Operational_Api, value :
    TempControlSoftwareSystem.SetPoint_i): Unit = {
    Contract(
      Requires(
        // BEGIN_COMPUTE_REQUIRES_setPoint
        api.fanCmd.isEmpty,
        api.setPoint == value
        // END_COMPUTE REQUIRES_setPoint
      ),
      Modifies(
        api,
        // BEGIN_COMPUTE_MODIFIES_setPoint
        currentSetPoint,
        currentFanState,
        latestTemp
        // END_COMPUTE MODIFIES_setPoint
      ),
      Ensures(
        // BEGIN_COMPUTE_ENSURES_setPoint
        // guarantee TC_Req_01
20        //   If the current temperature is less than the set point, then the fan
              state shall be Off.
21        (latestTemp.degrees < currentSetPoint.low.degrees) ->: (currentFanState ==
              CoolingFan.FanCmd.Off),
22        // guarantee TC_Req_02
23        //   If the current temperature is greater than the set point,
24        //   then the fan state shall be On.
25        (latestTemp.degrees > currentSetPoint.high.degrees) ->: (currentFanState ==
              CoolingFan.FanCmd.On),
26        // guarantee TC_Req_03
27        //   If the current temperature is greater than or equal to the
28        //   current low set point and less than or equal to the current high set
              point,
29        //   then the current fan state is maintained.
30        (latestTemp.degrees >= currentSetPoint.low.degrees & latestTemp.degrees <=
              currentSetPoint.high.degrees) ->: (currentFanState == In(currentFanState
              )),
31        // END_COMPUTE ENSURES_setPoint
32      )
33    )
34    api.logInfo("received setPoint")
35    currentSetPoint = value
36    perform_fan_control(api)
37  }
```

For space considerations, we do not include all contracts specified for the handler shown in the listing above.

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 COMPUTE 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.

The assume and guarantee descriptor strings and identifiers are translated into comments as seen on, e.g., line 19–20.

The handler method stubs are generated by HAMR and the developer fleshes out the application code. As the developer codes, Logika verification occurs behind the scenes and displays verification results to the developer.

8 Integration Constraints

Integration constraints are the invariants on the values flowing across ports. When specifying the integration constraints, we identify the requirements and assumptions that refer to a single port. In this section, we describe the syntax and semantics of integration constraints and refer to our temperature control example.

8.1 General Concepts

GCL integration contracts enable developers to specify constraints on values flowing in and out a component C's ports

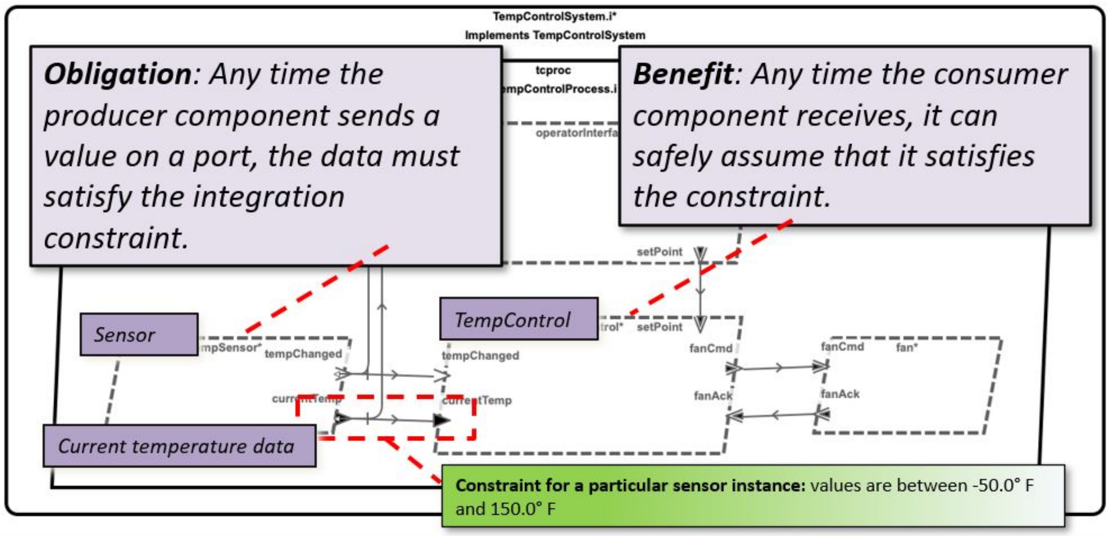

TempControlSystem.i*
Implements TempControlSystem
Obligation: Any time the producer component sends a value on a port, the data must satisfy the integration constraint.
tcproc
mpControlProcess.i
operatorInterfa
Benefit: Any time the consumer component receives, it can safely assume that it satisfies the constraint.
setPoint
Sensor
mpSensor*
tempChanged
currentTemp
Current temperature data
TempControl
trol*
setPoint
tempChanged
currentTemp
fanCmd
fanAck
fanCmd
fan*
fanAck
Constraint for a particular sensor instance: values are between -50.0° F and 150.0° F

Figure 11: GCL integration constraint concepts.

As shown in Figure 11, the `Sensor` component must ensure that the `currentTemp` data satisfies the integration constraint specified on the datatype. As a consumer of that data, the `TempControl` component can safely assume that the receiving data satisfies the constraint. This constraint is what we refer to as an integration constraint. In short, an integration constraint specifies the assumptions of an input port and the guarantees of an output port.

When an integration constraint is specified for a input port, the intent is that all values flowing into the port must satisfy the constraint. Similarly, for an integration constraint specified on an output port, all values flowing out of that port must satisfy its constraint.

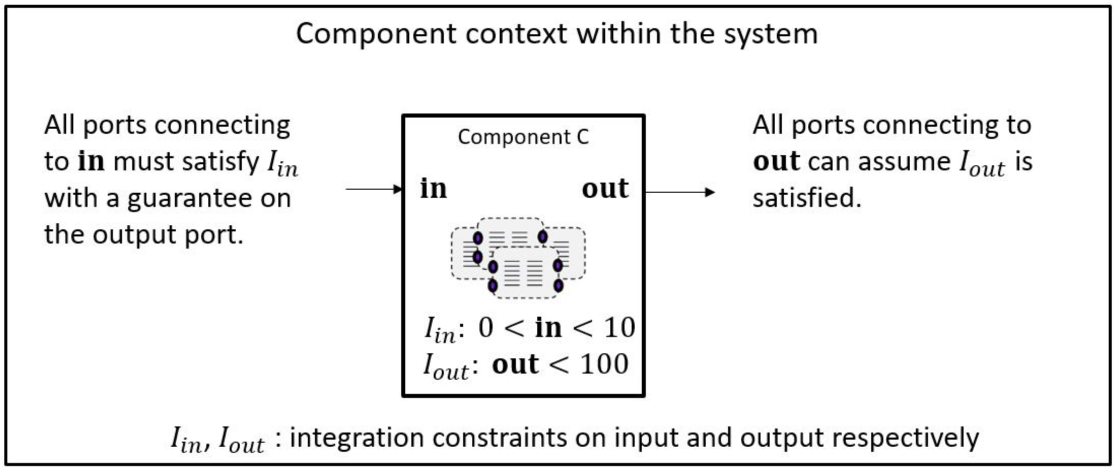

Component context within the system
All ports connecting to **in** must satisfy I_{in} with a guarantee on the output port.
Component C
in
out
All ports connecting to **out** can assume I_{out} is satisfied.
I_{in}: $0 < \textbf{in} < 10$
I_{out}: $\textbf{out} < 100$
I_{in}, I_{out} : integration constraints on input and output respectively

Figure 12: Integration constraints in system context and component context.

As shown in Figure 12, these contracts limit what other ports the component can connect to when it is integrated into a system context. For example, all output ports on any component that connects to some input port `in` must produce values that satisfy the integration constraint specified for `in` (shown on left of Figure 12). Thus, the integration constraint can be seen both as a requirement on the component's context (i.e., on clients of the component) and an assumption that the component makes about its context. An output port integration constraint can be seen as a requirement on the component to produce values on the output port that satisfy the output port's constraints (as shown in the figure as I_{out}).

If the component is verified to conform to its interface contract (which includes the integration constraints), then any other component that connects to an output port `out` can safely assume that values flowing out of `out` satisfy it's integration constraints. This is shown on the right of Figure 12. They may use this fact to establish that the integration contracts on their own input ports are satisfied.

To some extent, integration contracts, data invariants, and component entry point behavior contracts overlap in their potential uses. In contrast to a data invariant for type D which applies to all uses of values of D across the entire system, integration contracts enable a component to specify additional component/port-specific constraints that should hold for values flowing into or out of C across a port of type D.

In contrast to component behavior contracts which capture the functional behavior of a component (i.e., relationship between input and output port values), each integration constraint only pertains to values flowing in or out of a single port. The interpretation of integration constraints does not depend on the dispatch protocol of the component (e.g., periodic or sporadic) nor a specify entry point of the component. Thus, integration constraints can be understood as invariants on port values that hold across all entry points of the component.

The invariant takes the same syntactic form and has the same intuition for both data and event data ports (and this has the benefit of allowing developers to change a port's category, e.g., from data to event data) without having to rework the constraint specification. For a data port, all data values put into or taken out of the port must satisfy an associated integration contract. For an event data port, all payload values of messages that are queued in the port for sending and receiving must satisfy the integration constraint. Since event ports carry no values, integration constraints are not applicable for event ports.

Integration contracts are often one of the first types of constraints to be introduced by developers when constructing their AADL models. To use them, developers don't have to understand the subtleties of AADL dispatch semantics, entry points, event data port queuing, etc.; rather, they simply focus capturing basic constraints on the inputs and outputs of a component that go beyond types. This enables development teams to focus on basic aspects of the "connectability" or "compatibility" of components.

Verification of integration constraints can help control integration of components by guaranteeing that basic assumptions that a component makes its usage are enforced as this system is put together. This is especially important for multi-organizational development, and it is key to realizing the concept of virtual integration as represented by concepts such as ACVIP.

8.2 Model-level Syntax and Examples

The temperature control system was described in Section 3. We show examples using both the `TempControlPeriodic` and the `TempSensor` threads to illustrate integration constraints. When specifying the integration constraints, we identify the requirements and assumptions that refer to a single port. Recall that integration constraints do not relate inputs to outputs, but rather state an invariant over a single port value.

There are restrictions on what assumption and guarantee expressions may refer to within entry points:

- Assumptions can only refer to a single input port.

- Guarantees can only refer to a single output port.

For the `TempControlPeriodic` thread, we have an assumption that the thread can make about the values it reads on the `currentTemp` input port. The constraints are shown in Figure 13.

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

Figure 13: Integration constraints for the temperature control thread

As an example showing an integration constraint on an output port, we use the temperature sensor component. The `TempSensor` AADL definition is shown in the listing below.

```
thread TempSensor
    features
      currentTemp: out data port Temperature.i;
      tempChanged: out event port;
        flows
      cto: flow source currentTemp;
      tco: flow source tempChanged;
    properties
      Dispatch_Protocol => Periodic;
      Period => 1 sec;
end TempSensor;
```

There are two output ports on the temperature sensor thread. The event port contains no values and so no integration constraint can be defined for the `tempChanged` port. The `currentTemp` port, however, has the associated requirement *S-Req-02* as stated in Section 3.1. This is a constraint on a single output port that must always hold for the sensor component. Thus, we write it as an integration constraint. This is shown in the listing below.

```
integration
  guarantee tempRange "Sensor temperature range.":
    (currentTemp.degrees >= (f32"-50.0")) && (currentTemp.degrees <= f32"150.0");
```

The integration constraint on the `currentTemp` field specifies the operating temperature of the sensor as per the requirements.

Code level verification information will be added to this document as further support for integration constraints is provided.

Acronyms

AADL	Architecture Analysis and Design Language
AIR	AADL Intermediate Representation
API	Application Programming Interface
CAMET	Curated Access to Model-based Engineering Tools
CASE	Cyber Assured System Engineering
DARPA	Defense Advanced Research Projects Agency
GCL	GUMBO Contract Language
GUMBO	Grand Unified Modeling of Behavioral Operators
HAMR	High Assurance Modeling and Rapid engineering framework
IVE	Integrated Verification Environment
KSU	Kansas State University
OSATE	Open Source AADL Tool Environment
RTS	Run-Time Services
SAVI	System Architecture Virtual Integration

References

[1] Department of Defense Architecture Framework (DoDAF) website. https://dodcio.defense.gov/library/dod-architecture-framework/

[2] Future Airborne Capability Environment (FACE) website. https://www.opengroup.org/face

[3] GCL case studies. https://github.com/santoslab/hilt22-case-studies/

[4] Sireum GCL website. https://github.com/sireum/aadl-gumbo/blob/master/org.sireum.aadl.gumbo/src/org/sireum/aadl/gumbo/Gumbo.xtext

[5] Sireum HAMR course website. http://s21.highassurance.santoslab.org/lectures.html#lectures

[6] Sireum HAMR website. http://hamr.sireum.org/index.html

[7] Sireum Logika website. https://logika.v3.sireum.org/index.html

[8] sel4 microkernel (2015), sel4.systems/

[9] Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: a versatile and industrial-strength SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)

[10] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: International Conference on Computer Aided Verification. pp. 171–177. Springer (2011)

[11] Belt, J., Hatcliff, J., Robby, Shackleton, J., Carciofini, J., Carpenter, T., Mercer, E., Amundson, I., Babar, J., Cofer, D., Hardin, D., Hoech, K., Slind, K., Kuz, I., Mcleod, K.: Model-driven development for the seL4 microkernel using the HAMR framework. Journal of Systems Architecture p. (to appear) (2022)

[12] C, S.A.R.: Architecture analysis and design language (AADL) (2017)

[13] Conchon, S., Coquereau, A., Iguernlala, M., Mebsout, A.: Alt-ergo 2.2. In: SMT Workshop: International Workshop on Satisfiability Modulo Theories (2018)

[14] Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2013)

[15] Hatcliff, J., Belt, J., Robby, Carpenter, T.: HAMR: an AADL multi-platform code generation toolset. In: Margaria, T., Steffen, B. (eds.) 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, pp. 274–295. Springer (2021). https://doi.org/10.1007/978-3-030-89159-6_18, https://doi.org/10.1007/978-3-030-89159-6_18

[16] Hatcliff, J., Hugues, J., Stewart, D., Wrage, L.: 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) (2022)

[17] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: Compcert-a formally verified optimizing compiler. In: ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress (2016)

[18] Moura, L.d., Bjørner, N.: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)

[19] Richters, M., et al.: A precise approach to validating UML models and OCL constraints. Citeseer (2002)

[20] Robby, Hatcliff, J.: Slang: The sireum programming language. In: International Symposium on Leveraging Applications of Formal Methods. pp. 253–273. Springer (2021)

[21] Shames, P., Skipper, J.: Toward a framework for modeling space systems architectures. In: SpaceOps 2006 Conference. p. 5581 (2006)

[22] Ward, D.T., Helton, S.B.: Estimating Return on Investment for SAVI (a Model-Based Virtual Integration Process. In: SAE International Nournal of Aerospace (November 2011)