

Executable Interrupt Specifications: Capturing Interrupt Architecture as Machine-Readable Design Intent

Sindhu Amireddy
SoC Architecture
Microsoft Corporation
Austin, Texas, USA
samireddy@microsoft.com

Yi-Hua (Sammy) Shen
Silicon & Firmware Engineering
Microsoft Corporation
Mountain View, California, USA
yihshen@microsoft.com

Kiranmayee Sripathineni
Firmware Architecture
Microsoft Corporation
Santa Clara, California, USA
kiranmayees@microsoft.com

Aditya Suresh Prabhu
Firmware Engineering
Microsoft Corporation
Santa Clara, California, USA
adiprabhu@microsoft.com

Abstract— Modern chiplet-based systems-on-chip contain hundreds of interrupt sources distributed across IPs spanning multiple chiplets. Interrupts may traverse multiple layers of aggregation structures and routing elements, and interrupt controllers before reaching their designated servicing agent. Interrupt architectures are traditionally documented through spreadsheets, diagrams, textual specifications, firmware definitions, and verification collateral. Maintaining consistency across these independently authored artifacts becomes increasingly difficult as source definitions and system-integration decisions evolve. Because these artifacts encode overlapping architectural information, changes to an interrupt definition must be propagated across multiple representations, increasing the risk of divergence between downstream consumers and the architectural source.

This paper presents an executable interrupt specification methodology that captures architectural intent in a hierarchical, machine-readable form. The methodology separates reusable source semantics from SoC-specific integration. YAML-based IP-level specifications describe interrupt semantics, attributes, ownership, and handling requirements, while SoC-level specifications represent connectivity, routing, aggregation, interrupt-controller mappings, and servicing ownership across chiplets. An automated Single Source of Truth framework elaborates and validates these distributed specifications into a unified system model. Validation is performed over both local definitions and elaborated topology, enabling detection of unresolved references, incomplete paths, mapping conflicts, and missing ownership. The resulting model represents interrupt sources, aggregation stages, routing elements, controllers, identifiers, servicing owners, and their end-to-end dependencies.

Architecture views, firmware collateral, verification artifacts, and engineering reports are derived from the validated model rather than maintained as independent sources of architectural information. The resulting approach improves consistency, traceability, and reviewability across architecture, firmware, and verification workflows. It also enables systematic architectural queries and change-impact analysis without requiring engineers to manually correlate multiple representations of the same architectural information. Because the architectural data are structured and machine-readable, the framework provides a

foundation for AI-assisted specification authoring, design review, debugging, and knowledge retrieval. Although demonstrated for interrupt architectures, the methodology can be extended to other system-level specification domains, including Reliability, Availability, and Serviceability definitions and Performance Monitoring Unit infrastructure.

Keywords—*interrupt architecture, executable specifications, single source of truth, machine-readable specifications, system-on-chip (SoC), chiplet architectures, architectural modeling*

I. INTRODUCTION

Interrupts are a fundamental communication mechanism in a system-on-chip (SoC), allowing hardware components to report conditions that require software or firmware attention. These conditions include normal operational events, such as command completion and threshold notifications, as well as error and reliability events that may require immediate system-level handling. As server SoCs increase in scale and adopt disaggregated implementations, interrupt architecture must coordinate an expanding set of sources, routing structures, controllers, and servicing agents.

Chiplet-based designs make this coordination substantially more complex. In a monolithic SoC, interrupt sources and controllers typically reside within a common die boundary, and many routing relationships can be described locally. In a disaggregated system, an interrupt may originate in one chiplet, pass through local and system-level aggregation structures, cross die-to-die connectivity, receive one or more identifier mappings, and terminate at a controller associated with a different processing or management domain. Interrupt definition, transport, and servicing therefore become properties of the complete system rather than of an individual IP block.

Figure 1 illustrates the evolution from centralized interrupt handling in monolithic SoCs to distributed routing, aggregation, and servicing in chiplet-based systems.

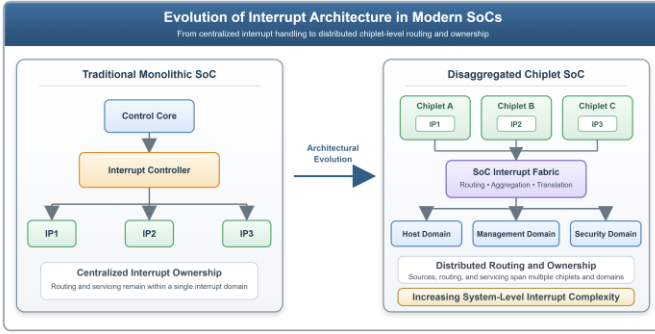


Fig. 1. Evolution from centralized interrupt handling in a monolithic SoC to distributed interrupt architecture in a chiplet-based system.

Interrupt architecture is commonly documented through spreadsheets, block diagrams, textual specifications, firmware definitions, and verification models. Each artifact serves a specific engineering audience, but several artifacts may independently represent the same source attributes, routing decisions, identifiers, and ownership assignments. When the architecture changes, corresponding updates must be propagated across these representations. A missing or inconsistent update can create ambiguity during implementation, integration, verification, or design review.

The fundamental challenge is therefore not simply how to document a large interrupt map. The challenge is how to capture architectural intent once, validate it systematically, and derive the representations required by different engineering workflows. A useful specification must remain understandable to architects while also being sufficiently structured for automated analysis and artifact generation. It must represent both IP-local interrupt semantics and system-level relationships spanning chiplets, aggregators, controllers, and servicing agents.

This paper introduces a relationship-centric executable specification methodology for SoC interrupt architecture. Unlike document-centric interrupt maps or record-level schemas, the methodology separates reusable source semantics from SoC-specific integration and elaborates distributed definitions into resolved end-to-end interrupt paths. Validation is performed over the resulting system topology, enabling detection of unresolved references, incomplete paths, mapping conflicts, and missing ownership. The validated model supports deterministic propagation of architectural information to downstream engineering representations, reducing manual synchronization and re-entry across independently maintained artifacts. It also preserves traceability from generated artifacts to their authoritative specification entries.

The central contribution is not a new serialization format or interrupt-controller description, but a validated architectural dependency model that supports validation, traceability, derived engineering views, and change-impact analysis.

The principal contributions of this paper are:

1. **A hierarchical specification model** that separates reusable IP-level interrupt intent from SoC-level integration and routing.

2. **An SSOT elaboration and validation methodology** that constructs a unified representation from distributed specifications.
3. **A derived-artifact workflow** for generating discipline-specific engineering views from the validated system model.
4. **A structured architectural knowledge foundation** that supports queries, change-impact analysis, and AI-assisted engineering workflows.

The remainder of this paper is organized as follows. Section II examines the limitations of traditional interrupt architecture specifications and the challenges of maintaining consistency across architectural representations. Section III introduces the executable interrupt specification framework, including the hierarchical specification model, system-level model construction, and validation methodology. Section IV describes the application of the methodology and its use in generating engineering views, architectural analysis, and change-impact assessment. Section V discusses AI-assisted engineering workflows enabled by the structured architecture model. Section VI considers the applicability of the approach to other SoC specification domains. Section VII concludes the paper and summarizes key observations from deployment of the methodology.

II. LIMITATIONS OF TRADITIONAL INTERRUPT ARCHITECTURE SPECIFICATIONS

Modern interrupt architectures place increasing demands on specification methodologies. As systems scale and become more distributed, interrupt behavior is defined not only by individual interrupt sources but also by the relationships among sources, routing infrastructure, interrupt controllers, and servicing agents. Capturing these relationships accurately, maintaining consistency across engineering artifacts, and managing architectural change throughout the development lifecycle have become increasingly challenging. The following subsections examine key limitations of traditional interrupt architecture specifications and motivate the need for a more structured and executable approach.

A. Interrupt Relationships Are Not First-Class Specification Elements

Traditional interrupt specifications are primarily organized around interrupt definitions. Individual interrupt sources, identifiers, descriptions, and attributes are typically documented explicitly, while many of the relationships that determine system behavior are represented indirectly through diagrams, tables, integration documents, or institutional knowledge.

However, modern interrupt architectures derive much of their behavior from these relationships. Routing paths, aggregation hierarchies, controller assignments, servicing ownership, and cross-chiplet connectivity collectively determine how interrupts propagate through the system and ultimately how they are handled.

As systems grow in scale, the architectural significance of these relationships often exceeds that of the individual interrupt definitions themselves. Yet traditional specifications generally treat these relationships as supporting information rather than as

explicitly represented architectural entities. This makes it difficult to analyze, validate, and maintain interrupt behavior as a coherent system-level construct.

B. Interrupt Definition and System Integration Evolve Independently

IP-level interrupt definitions are often created long before system integration decisions are finalized. During platform integration, additional architectural information is introduced, including routing paths, aggregation behavior, controller mappings, ownership assignments, and servicing policies.

These system-level decisions frequently evolve independently of the originating interrupt definition. A change in controller partitioning, firmware ownership, or routing strategy may alter interrupt behavior without modifying the source definition itself. Consequently, interrupt architecture is distributed across multiple layers of specification that must remain consistent over time.

Traditional specification approaches provide limited mechanisms for maintaining these relationships as architectural decisions evolve, increasing the effort required to preserve consistency between IP-level and system-level views.

C. Maintaining Consistency Across Representations

Interrupt architecture information is consumed by multiple engineering disciplines throughout product development. Architecture teams require specifications and connectivity views, firmware teams require software-visible definitions and configuration information, and verification teams require models that represent expected system behavior. Additional representations are often created for architecture reviews, design documentation, bring-up activities, and debug workflows.

Although these artifacts serve different purposes, they frequently describe overlapping aspects of the same interrupt architecture. Interrupt ownership, routing relationships, aggregation behavior, controller mappings, servicing requirements, and interrupt classifications may appear in multiple representations simultaneously.

The challenge is not the existence of multiple views, but the effort required to keep them aligned. Architectural changes that appear localized often require updates across numerous downstream artifacts. As architectures evolve, ensuring consistency across these representations becomes increasingly difficult, and substantial engineering effort may be devoted to validating agreement among them.

D. Beyond Documentation-Centric Specifications

Traditional interrupt specifications are primarily designed to capture and communicate architectural information. Documents, spreadsheets, diagrams, and review collateral remain effective mechanisms for communicating design decisions, but they provide limited support for managing the evolving relationships that define modern interrupt architectures. As systems scale, engineers must trace ownership across management domains, evaluate routing changes, analyze aggregation dependencies, and assess the effects of architectural modifications on firmware and verification environments. These activities require interrupt architecture to be examined as an

interconnected system rather than as a collection of independently documented elements.

The limitation is therefore not a lack of documentation, but the absence of an authoritative, structured representation of interrupt architecture that can be systematically queried, validated, and transformed. Consequently, architectural understanding often depends on manually correlating information distributed across multiple artifacts. Addressing this limitation requires specifications to evolve from static descriptions into active representations of architectural knowledge. The following section introduces an executable interrupt specification methodology that provides such a representation through a machine-readable Single Source of Truth framework.

III. EXECUTABLE INTERRUPT SPECIFICATION FRAMEWORK

The proposed methodology represents interrupt architecture using hierarchical, machine-readable specifications and an elaborated system model. In this paper, an executable specification is a machine-readable architectural representation with defined semantics that supports deterministic elaboration, validation, querying, and transformation into downstream engineering artifacts. The framework implements this methodology by processing the specifications through normalization, elaboration, and validation. The resulting system model serves as the central architectural representation, while the specifications provide the structured inputs from which the model is derived.

The system model represents interrupt sources, aggregation stages, routing elements, controllers, and servicing agents as architectural entities. Connections among these entities capture interrupt propagation paths, enabling end-to-end traceability from a source definition to its system-level destination and servicing owner.

A. Executable Interrupt Architecture Model

An interrupt source is represented by its semantics and attributes together with the system relationships that determine its delivery and servicing. Source-level information includes the event definition, classification, severity, and handling requirements. Integration information includes aggregation behavior, routing, controller and identifier mappings, and servicing ownership.

As shown in Fig. 2, these two classes of information are combined in an end-to-end path:

Source→Aggregator→Router→Controller→Servicing Agent

The precise path may vary by source and system topology. The model does not require every interrupt to traverse every entity type. Instead, it records the applicable entities and directed relationships for each instantiated path.

Representing these relationships explicitly enables the framework to distinguish between source semantics and integration decisions. For example, severity may be intrinsic to an event, whereas controller assignment and servicing

ownership may depend on the target SoC. Both remain associated with the same instantiated interrupt without embedding platform-specific information in the reusable source definition.

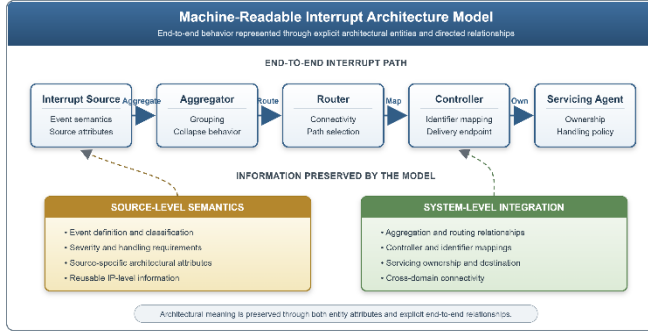


Fig. 2. Machine-readable interrupt architecture model combining source-level semantics with system-level integration relationships.

B. Hierarchical Specification Structure

The framework organizes specification content into two complementary layers. IP-level specifications define reusable interrupt sources. SoC-level specifications instantiate those definitions and add the relationships required by a particular system.

An IP-level specification records information owned by the source IP, including:

1. source name and description;
2. event classification and severity;
3. source-specific attributes; and
4. handling requirements that remain valid across instantiations.

A SoC-level specification records information established during integration, including:

1. IP instances and system connectivity;
2. aggregation and routing relationships;
3. interrupt-controller and identifier mappings; and
4. servicing destinations and ownership assignments.

The two layers are linked through references rather than duplication. An SoC specification identifies an instantiated IP interrupt and adds integration context without redefining the source semantics. This structure preserves traceability between a system interrupt and its originating IP definition.

The separation also defines the scope of a change. A change to an event classification is made in the IP-level specification. A change to a route, controller assignment, or servicing owner is made in the SoC-level specification. The elaborated model incorporates both changes and evaluates the affected relationships.

A simplified, nonproprietary example is shown below. The example illustrates the separation of concerns and is not intended to prescribe a complete schema.

Reusable IP-level definition

```
interrupt:
  name: fatal_error
  classification: ras
  severity: fatal
```

SoC-level integration

```
interrupt_instance:
  source: subsystem0.fatal_error
  route:
    - local_aggregator
    - system_router
    - management_controller
  servicing_owner: system_management_core
```

The serialization format provides a human-reviewable representation, but the model semantics do not depend on YAML-specific behavior. The methodology depends on defined entities, attributes, references, and relationships that can be interpreted and validated consistently.

C. Model Construction and Validation

The distributed specifications are processed through normalization, elaboration, and validation, as illustrated in Fig. 3.

During normalization, the framework interprets the input structures, checks required fields and data types, and resolves references to declared definitions. This stage converts independently authored inputs into a common internal representation.

During elaboration, the framework combines reusable IP definitions with SoC integration data. It resolves instances, constructs aggregation and routing relationships, applies controller mappings, and associates interrupt destinations with servicing owners. The resulting graph represents the interrupt topology after system integration rather than a collection of isolated specification entries.

Validation is performed on both local entries and elaborated relationships. Representative validation classes include:

1. **Definition integrity:** required attributes, valid types, and unique identifiers.
2. **Reference integrity:** valid IP instances, sources, destinations, and mappings.
3. **Connectivity completeness:** resolved paths between applicable sources and destinations.
4. **Topology consistency:** valid aggregation and routing relationships.
5. **Mapping consistency:** complete controller and identifier assignments.
6. **Ownership completeness:** defined servicing ownership where required.

These validation classes are enforced through architectural invariants evaluated on the elaborated model. Representative invariants include:

- Every instantiated interrupt source must resolve to at least one valid delivery path.
- Every interrupt requiring service must resolve to a servicing owner.
- Each software-visible controller identifier must be unique within its declared namespace.
- Routing relationships must not contain cycles unless explicitly modeled as legal feedback paths.
- Aggregation must preserve properties required for downstream routing and servicing.

Diagnostics identify the specification entry or relationship associated with failure. This allows the source information to be corrected before downstream artifacts are derived. The validated output becomes the authoritative interrupt architecture model used by subsequent workflows.

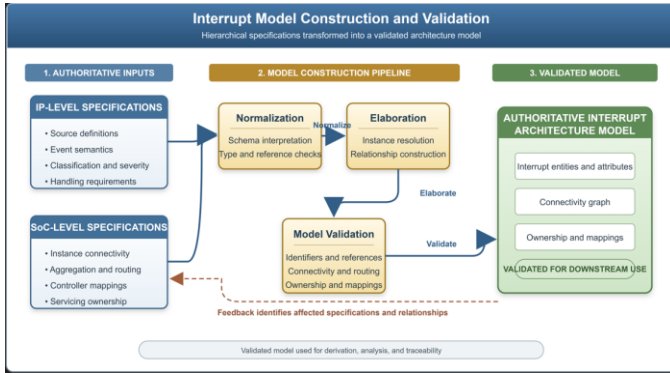


Fig. 3. Construction and validation of the interrupt architecture model from hierarchical specifications.

D. Single Source of Truth and Model Consumption

The validated model supports two principal uses: deterministic derivation of engineering views and analysis of architectural relationships. Figure 4 summarizes these uses.

Engineering disciplines consume different subsets and representations of the model. Representative outputs include:

1. interrupt maps and connectivity reports;
2. routing and ownership views;
3. software-visible definitions and configuration data;
4. verification mappings and checking collateral; and
5. architecture-review summaries.

These outputs remain specialized for their respective workflows, but common interrupt information originates from the same validated representation. The model therefore functions as a **Single Source of Truth (SSOT)** without requiring every discipline to consume an identical file or format.

The model also supports relationship-oriented analysis. Representative operations include tracing a source to its controller and servicing owner, identifying interrupts assigned to a particular agent, locating paths that cross a chiplet boundary, and determining which model elements depend on a proposed mapping change.

Both derivation and analysis preserve traceability to the originating model entities and relationships. Hierarchical references further connect these elements to the corresponding IP-level definitions and SoC-level integration entries. This allows generated values, query results, and affected dependencies to be traced to their authoritative specification sources.

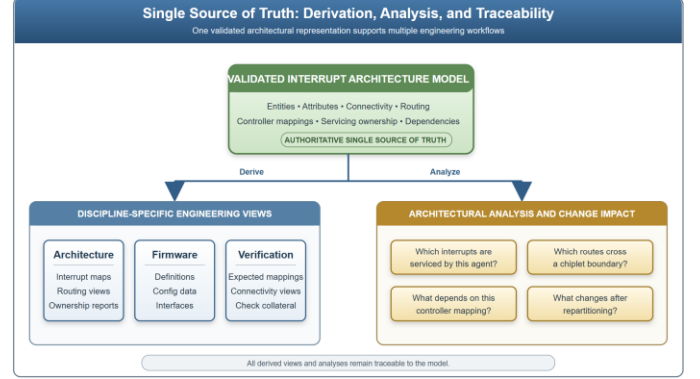


Fig. 4. Derivation of engineering views and architectural analysis from the validated interrupt architecture model.

The framework was realized in an internal implementation and applied to interrupt-specification workflows for complex SoC architectures. The following section describes the resulting engineering artifacts, representative analysis capabilities, and observed workflow improvements using generalized examples that omit product-specific information.

IV. APPLICATION OF THE METHODOLOGY

The executable interrupt specification methodology was implemented and evaluated using specifications representative of a modern chiplet-based SoC. Specifications defined across multiple levels of hierarchy were integrated into a unified interrupt architecture model, enabling generation of system-level engineering representations and architectural analysis.

A. SoC-Level Interrupt Topology

One practical outcome of the framework is the automatic construction of a system-level interrupt topology from distributed interrupt specifications. The resulting representation captures interrupt sources, aggregation hierarchies, routing relationships, controller mappings, and servicing ownership across the SoC. Rather than manually constructing and maintaining separate interrupt trees, the integrated topology is derived directly from validated specifications.

The generated topology supports multiple engineering perspectives, including:

- source-to-controller mappings;
- ownership-based interrupt views;
- processor-specific interrupt trees; and
- routing paths through aggregation infrastructure.

TABLE I ILLUSTRATES A REPRESENTATIVE TOPOLOGY.

Source	Route	Controller	Servicing Owner
--------	-------	------------	-----------------

Fatal Error	Local Aggregator → System Router	Management Controller	Management firmware
Security Event	Security Aggregator → Secure Router	Secure Controller	Security Firmware
Completion Event	Direct Route	Host Controller	Host firmware

This table is illustrative and does not correspond to a specific implementation.

B. Generated Engineering Collateral

Beyond topology visualization, the framework generates collateral for downstream firmware and verification workflows, including interrupt maps, HTML-rendered architectural views, and software-oriented representations. Each output is tailored to its consumer while retaining traceability to the authoritative source definitions and integration relationships.

C. Evaluation Results

The methodology was evaluated using interrupt specifications representative of a complex chiplet-based SoC. The reported figures are generalized to characterize evaluation scale without exposing implementation-specific details. The evaluated inputs comprised more than one hundred specification files and more than five hundred reusable block definitions. Elaboration produced an integrated topology containing hundreds of aggregation, routing, and internal topology elements, with more than three thousand resolved interrupt paths. The resulting model was used to exercise hierarchical elaboration, architectural validation, derivation of engineering views, traceability, and propagation of representative architectural changes.

TABLE II SUMMARY OF EVALUATION RESULTS

Evaluation dimension	Evaluation result
Specification scale	More than one hundred input files and more than five hundred reusable block definitions
Elaborated topology	Hundreds of aggregation, routing, and internal topology elements
Resolved connectivity	More than three thousand resolved interrupt paths
Architectural validation	Checks were exercised for definition integrity, reference integrity, connectivity completeness, topology consistency, mapping consistency, and ownership completeness
Derived engineering views	Multiple architecture, firmware, verification, and review-oriented representations derived from a common model
Change propagation	Representative source-level and system-integration changes were incorporated through model re-elaboration, with dependent relationships and representations identified from the updated model
Traceability and impact analysis	Dependencies between source definitions, instantiated relationships, and derived representations were preserved for change-impact analysis

Representative changes were applied at the specification layer to exercise both source-level properties and system-level

integration relationships. The framework re-elaborated the affected relationships, applied architectural validation, and identified dependent representations from the common model.

These results demonstrate a central property of the SSOT methodology: architectural intent can be maintained at its authoritative source and propagated systematically through the elaborated system model to dependent representations. This approach reduces reliance on manual synchronization and repeated re-entry of architectural information across independently maintained artifacts.

D. Architectural Queries and Dependency Analysis

The architecture model enables direct evaluation of interrupt relationships that are otherwise distributed across multiple artifacts. Because interrupt routes, ownership assignments, controller mappings, and aggregation relationships are represented explicitly, the model supports topology-oriented analysis.

Representative analyses include:

- identifying all interrupts serviced by a selected firmware agent;
- tracing interrupt paths through aggregation and routing infrastructure;
- locating interrupts associated with a specific controller;
- examining ownership distributions across management domains; and
- evaluating dependencies affected by routing or ownership changes.

For example, a controller reassignment can be analyzed by identifying the affected interrupt paths, associated ownership relationships, and downstream views derived from those mappings. Since these dependencies are represented within the architecture model, the impact assessment is performed on the elaborated interrupt topology rather than reconstructed manually from separate artifacts.

V. AI-ASSISTED ENGINEERING WORKFLOWS

The value of AI in this context does not depend on an interrupt-specific reasoning mechanism. Instead, the structured architecture model provides grounded context for AI-assisted retrieval, specification review, and change exploration. Because model entities, attributes, and relationships remain traceable to authoritative specification entries, AI-generated observations can be linked to the architectural information on which they are based.

A. Architectural Knowledge Retrieval

For knowledge retrieval, the model supplies explicit context about interrupt entities, attributes, and relationships. This context can support questions such as:

- Which interrupts are serviced by a specific management agent?

- Which interrupt paths traverse a given routing structure?
- Which interrupts are classified as fatal events?
- Which interrupt sources are connected to a selected controller?

Responses can be grounded in the relevant source definitions, instantiated paths, controller mappings, and ownership assignments.

B. Specification Authoring and Review Assistance

Structured interrupt data can support specification authoring and review by providing reusable context from existing definitions, ownership policies, and routing patterns. Potential applications include:

- identifying specification inconsistencies;
- highlighting missing ownership assignments;
- detecting incomplete routing information; and
- summarizing architectural relationships for review preparation.

AI-generated observations can be associated with the originating model entities and relationships, allowing engineers to inspect the supporting architectural context.

C. Architecture Debug and Change Analysis

For debugging, AI-assisted workflows can use the connected model to gather the source definitions, routing paths, controller mappings, and servicing assignments relevant to an interrupt event. For proposed architectural changes, AI can help engineers formulate questions and summarize affected dependencies, while deterministic model queries and validation remain responsible for establishing the underlying relationships and constraint results.

VI. EXTENDING BEYOND INTERRUPT ARCHITECTURES

Although this work focuses on interrupt architecture, the methodology is applicable to a broader class of SoC architectural specifications. The key requirement is the existence of architectural entities, attributes, and relationships that must remain consistent across multiple engineering workflows.

Interrupt architecture represents one example of this problem. Similar challenges exist in Reliability, Availability, and Serviceability (RAS) infrastructure, Performance Monitoring Unit (PMU) definitions, telemetry frameworks, and other architecture domains where information is distributed across specifications, firmware implementations, verification environments, and review

collateral. The same principles used to represent interrupt connectivity, ownership, and routing relationships can be applied to these domains.

For example, RAS architectures contain relationships among error sources, classifications, reporting paths, and recovery actions. PMU infrastructures contain relationships among monitoring events, counters, routing mechanisms, and software-visible interfaces. In both cases, maintaining consistency across multiple engineering artifacts presents challenges similar to those addressed by the interrupt-specification framework.

More broadly, the methodology demonstrates how machine-readable architectural specifications can serve as authoritative representations of system-level design information. While the specific entities and relationships vary between domains, the concepts of model construction, validation, derivation, and traceability remain applicable.

VII. CONCLUSION

Modern chiplet-based SoCs require interrupt architecture to be managed as a connected system of sources, routing relationships, controller mappings, and servicing responsibilities. This paper presented an executable specification methodology that separates reusable interrupt semantics from SoC-specific integration and constructs a validated system model from distributed architectural inputs.

The implementation demonstrated hierarchical model construction, topology-based validation, deterministic derivation of engineering views, and traceability across representative architectural changes. By making interrupt entities and dependencies explicit, the methodology supports systematic analysis of system-level interrupt behavior and controlled propagation of architectural updates.

Although demonstrated for interrupt architecture, the principles of structured specification, elaboration, validation, derivation, and traceability apply to other relationship-intensive SoC domains. The resulting architectural knowledge models also provide grounded context for future automation and AI-assisted engineering workflows.

VIII. ACKNOWLEDGMENT

The authors gratefully acknowledge the many engineers whose contributions to interrupt architecture definition, specification development, framework implementation, and downstream adoption made this work possible. Their technical expertise, feedback, reviews, and continued engagement were instrumental in refining and evaluating the executable interrupt specification methodology within complex SoC development workflows.