Towards Applying Stream-Based Assumption/Guarantee Specifications for Runtime Verification

Florian Drux[1,*,†], Adrian Marin[1,*,†], Bernhard Rumpe[1,†] and David Schmalzing[1,†]

[1]RWTH Aachen University, Aachen, Germany

Abstract

In complex system and software engineering, specifying functional behavior early in the design process is essential for ensuring correctness and consistency across development artifacts. However, existing modeling languages, including SysML v2, are either too abstract or require committing to implementation details prematurely, limiting the ability to capture partial specifications. To overcome these limitations, we build on SysML v2 and propose a modeling notation for specifying assumption and guarantees using stream-processing functions. This notation enables formally specifying black-box behavior with a high degree of underspecification. We furthermore present and discuss ideas on how to use such specifications during simulation and testing to verify the functional behavior.

Keywords

Runtime Verification, Specification, SysML v2, Architecture Description Language

1. Introduction

Model-based systems engineering (MBSE) [1] aims to replace document-centered development with formal, analyzable models that capture system structure and behavior. A central goal in this transition is to formalize requirements early in order to reduce ambiguity, enable automated analysis, and improve overall system consistency. To avoid over-constraining the design during requirement formalization, engineers focus on defining the functional black-box behavior of components by describing observable inputs and outputs, thereby capturing what a system must do while abstracting from internal realization. Since system design and requirements are evolving, these descriptions must natively support underspecification, allowing requirements to be recorded accurately without forcing premature design decisions [2].

Existing modeling languages, including SysML v2 [3], provide strong support for structural modeling, requirements management, and state-based behavioral descriptions, but offer only limited means to express behavior with regard to input–output histories of components with explicit timing of events. Expressing input–output data flow underspecification is lacking in options as well. In particular, current approaches make it difficult to specify functional constraints over input-output histories of components in a way that is both abstract enough for early phases and precise enough for analysis and refinement [4].

Early in development, using underspecification one could, *e.g.,* specify that an elevator may not open its doors more often than the number of buttons pressed. This specification can later be refined during development, *e.g.,* the elevator may not open its doors unnecessarily, *i.e.,* not on floors that were not requested by passengers. This workflow heavily relies on the ability to express constraints over input-output streams as well as to refine underspecified requirements. Formalizing these requirements enables automated analysis, be it for static verification or runtime monitoring.

In this work, we build on the Focus [5] framework and propose a modeling notation for specifying stream-processing functions over component interfaces, which we embed into SysML v2. We further

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026

*Corresponding author.

†These authors contributed equally.

drux@se-rwth.de (F. Drux); marin@se-rwth.de (A. Marin); rumpe@se-rwth.de (B. Rumpe); schmalzing@se-rwth.de (D. Schmalzing)

0000-0002-6114-8776 (F. Drux); 0009-0006-5740-8439 (A. Marin); 0000-0002-2147-1966 (B. Rumpe); 0000-0003-2041-1475 (D. Schmalzing)



© 2026 Copyright © 2026 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

CEUR-WS.org/Vol-4338/ICMM-paper3.pdf

introduce a library of typed stream kinds with different timing abstractions [6], along with corresponding operators, enabling history-based, black-box specifications of components in terms of their input and output streams. Using this stream-based notation, we formulate Assumption/Guarantee (A/G) [7] and sketch how a computable subset of specifications can be transformed into executable monitors for runtime analysis. Our approach enables test- and monitor-driven verification of safety properties: realizations modeled in SysML v2 can be checked against history-based specifications by synthesizing monitors from A/G contracts and stream expressions.

Hence, our contributions are:

1. a stream library and its typing discipline for specifying input/output behavior,
2. an A/G notation embedded into SysML v2 for history-oriented black-box specifications, and
3. preliminary ideas on deriving and interpreting monitorable safety properties at runtime.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 introduces our stream library, including stream kinds, time abstractions, and operators for expressing constraints over input and output histories. Section 4 presents the embedding of this library into SysML v2 to specify A/Gs, and explains how to derive runtime monitors from such specifications. Section 5 discusses advantages and limitations, and Section 6 concludes the paper with a summary and directions for future work.

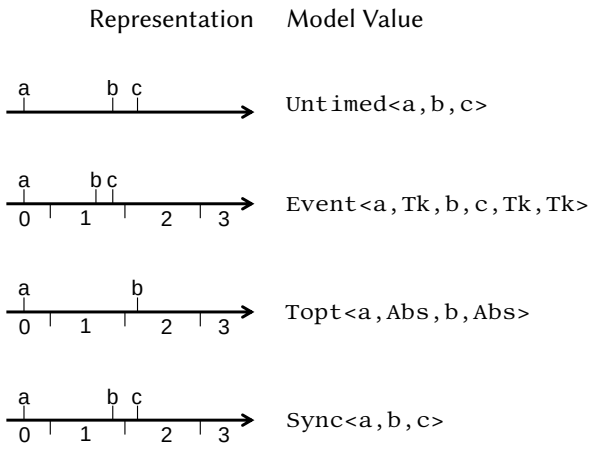
2. Related Work

Our approach applies the Focus [5] formalism in a practical modeling language for monitoring specifications. Most applications of Focus center on AutoFocus [8] and provide tooling for all development phases, with recent approaches focusing on property-based testing [9] and model checking [10] rather than runtime verification. Other formal verification approaches using Focus exist [11, 12]; these transform SysML v2 specifications into the theorem prover Isabelle. There is also an initial stream-library concept implemented in Isabelle [13] that serves as a backend for this verification framework. The advantage of this approach is that it can semi-automatically prove refinement between specifications at high levels of abstraction (*e.g.,* stream specifications), low levels (*e.g.,* automata), and levels in between. Our approach complements this theorem-prover-based work: by using the same specification language for modeling system constraints and the same mathematical framework for proving refinement, users can employ runtime verification where static verification fails.

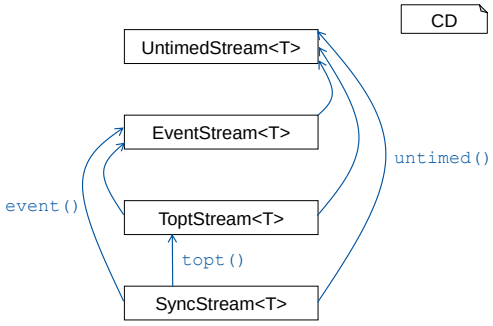
In the domain of runtime verification, some approaches perform static analysis on specifications to prune formally verifiable parts of properties, with the remaining parts checked through runtime verification [14]. In LARVA [15], runtime verification is performed using pre/postconditions and automata with timers and events. This differs from our approach because it is low-level and implementation-oriented, requiring a glass-box view, whereas we focus on black-box input-output traces.

Stream-based monitoring specifications are an active research area. Lola [16] and RTLola [17] stand out by having been successfully applied in DLR avionics case studies [18]. Lola employs monitorable stream specifications based on equations over input-output communication histories. Output streams are modeled as stream expressions, which are stream-processing functions. This approach is strictly time-synchronous, which can limit expressiveness and are not suitable for specifying reactive systems. RTLola extended the specification language with event-timed specifications. Both languages offer guarantees of predictable execution times and bounded memory usage by employing static analysis before monitoring. These guarantees restrict both languages to deterministic specifications. HStriver [19] provides a similar approach to RTLola by providing a runtime verification framework for real-time event-streams by applying a embedded domain-specific language (eDSL) into Haskell and borrowing flexible Haskell datatypes. The capability of expressing underspecification through non-determinism in the specification language and how it is handled by the framework when monitoring is not explicitly touched upon. In comparison, our language can additionally express non-deterministic properties, but in doing so we limit the monitorability of the specifications.

3. Stream Library



(a) A schematic representation of streams of the four different kinds, as well as an example of their concrete construction within a model; a, b, and c are messages, 0, 1, 2, and 3 denote the different chunks (slices) of time.



(b) The stream kinds form an order of specificity: `UntimedStream` is the most general kind, providing no time information whatsoever, while `SyncStream` is the most specific kind, with the most restrictive time constraints. Streams can be converted to less specific kinds by use of the corresponding conversion function.

Figure 1: Overview over the four stream kinds.

To define constraints ob the input and output behavior of components, we propose the following library of stream types and corresponding operators. We will define the different kinds and types of streams and describe how they are constructed. Afterward, we present examples of stream operations.

3.1. Stream Types

Essentially streams are finite or infinite sequences of messages with or without additional time information dependent on the stream kind. Each stream type consists of one stream kind out of `UntimedStream`, `EventStream`, `ToptStream`, and `SyncStream`, and the type of messages contained. The typing of a stream is used to consistently check the models using said streams. We write, *e.g.*, `EventStream<Mes>` as the type of an `EventStream` containing messages of type `Mes`. Figure 1a shows one example stream for each kind;

`UntimedStream` contains messages in a sequence with no further time information. This is the least constrained stream type, useful in cases where time information is either not required or not available. An example use case is input data of a system given before the system starts, *e.g.*, a sorted list of file names to process. While ordered, no time information is required for such a stream.

`EventStream` is comparable to an `UntimedStream` with additional time information; Messages are split into time slices by `Tks` (ticks), each time slice containing an arbitrary but finite amount of messages. This can be used to, *e.g.*, represent input/output of a system in relation to a clock, like button presses, internet data packets, etc.

`ToptStream` adds a further constraint to `EventStream`. Each time slice may contain no more than one message. An example usage is the abstracted representation of the communication of a Central Processing Unit (CPU). The CPU communicates with signals. By selecting the length of time slices to be the duration of these signals, each time slice either contains a signal, or is empty (`Abs`). While an `EventStream` contains more information than an `UntimedStream`, a `ToptStream` is directly representable by `EventStream`, and exists solely for ease of modelling and conciseness of data representation.

`SyncStream` is identical to `ToptStream`, except that each time slice must contain exactly one message. This is, similar to `ToptStream`, useful to represent signals and comparable message flows. `SyncStream` is used in case there is a guarantee that a message exists every time slice.

Different stream types fit different inputs/outputs of components. Generally speaking, the most specific applicable stream type should be used for a given channel. However, this can lead to components or configurations of components with pairwise different stream kinds, which complicate the writing of expressions over these streams. To solve this problem, we provide conversion functions between stream types. As some stream types can be directly represented by other stream types, expressions over different stream types can be written by first converting them to a common kind.
Their conversion functions are shown in Figure 1b. By removing all time information, any stream can be converted to an `UntimedStream`. A `SyncStream` can be converted to a `ToptStream`, as a `SyncStream` has exactly one message per time slice, while a `ToptStream` has at most one. Analogously, both `ToptStream` and `SyncStream` can be converted to `EventStream`, as they, essentially, are both `EventStreams` with an additional constraint each.

3.2. Stream Values

Streams can be compared to and combined with other streams. As such, it is necessary to create new streams, *e.g.,* to describe an expected message flow.

To create values of stream types, constructors are provided by the stream library. Streams can be created in place by listing the messages, and additional time information as required. The names of the constructors are the names of the stream kinds without "Stream", cf. Figure 1b. The message type can be either inferred from the context (in most cases), or explicitly provided within additional angle brackets.

When constructing stream values, time information needs to be provided explicitly for `EventStreams` and `ToptStreams`; For `EventStreams`, time slices are delimited by `Tks`. For `ToptStreams`, empty time slices are denoted by `Abs` (Absent). In the case of `UntimedStreams`, no time information exists, and for `SyncStreams`, time information is implicit, as each message is within its own time slice.

3.3. Stream Operators

To write constraints over streams, we propose a library of side-effect-free stream operators. Our goal in this section is not to present a concrete set of operators, including their concrete syntax, but rather to provide an intuitive overview on how to check properties of streams (*e.g.,* getting the length), as well as create modified versions of already existing streams (*e.g.,* via `map`). As streams are sequences of messages, many common list operators of languages such as, *e.g.,* Haskell [20] have direct correspondent in our library. Other operators however are designed to address the timing information of streams. A selected set of operators is shown in Table 1.

The *length* can only be returned for finite streams. Calling `#s` with `s` being an infinite stream is undefined. The length of a stream can be either known to be finite, known to be infinite, or unknown. For simple expressions, it is decidable whether the stream is finite, *e.g.,* `s.take(5)`, or infinite, *e.g.,* `s.infTimes()`. For more complex expressions, however, it can be undecidable, and `s.isInfiniteLen()` as well as `s.isFiniteLen()` return `false`. It is suggested to, most of the time, avoid writing expressions that rely on streams being finite for this reason.

Concatenation and taking substreams of streams are operations common with list operations akin to functional programming languages like Haskell [20]. As an instance of `UntimedStream` contains a sequence of messages without time information, while instances of other stream kinds do, the behavior differs for some operators. For example, `s.take(5)` returns the first five time slices of `s` for `EventStream`, `ToptStream`, and `SyncStream`, while it returns the first five messages of `s` for `UntimedStream`, and `SyncStream`.

Common functionality such as *`filter`*, *`map`*, *`scanl`*, and *`foldl`* (not depicted) behave as expected for Streams as they do for lists. However, for `EventStream` specifically, one can operate either on single messages, or on whole time slices. Thus, these operators have a "`Slice`" variant, *e.g.,* `mapSlice`, which applies the given function to each time slice.

Operation	Supported stream kinds	Description
`s == s2`	same kind	whether `s` and `s2` are pairwise identical
`s.nth(n)`	all	the n'th message (Untimed, Sync) the message of n'th time slice or nothing (Topt) the n'th time slice of (Event) (a finite `UntimedStream`)
`s.first()`	all	same as `s.nth(0)`
`#s`	all	length of `s`
`s.isInfiniteLen()`	all	whether `s` is *guaranteed* to be infinite
`s.isFiniteLen()`	all	whether `s` is *guaranteed* to be finite
`s ^^ s2`	same kind	concatenates `s` with `s2`
`s.times(n)`	all	`s` concatenated with itself n times
`s.infTimes`	all	`s` concatenated with itself infinitely often
`s.take(n)`	all	`s`, but only the first n elements (Untimed) `s`, but only the first n time slices (Event, Topt, Sync)
`s.dropMultiple(n)`	all	`s` without the first n elements (Untimed) `s` without the first n time slices (Event, Topt, Sync)
`s.dropFirst()`	all	same as `s.dropMultiple(1)`
`s.filter(p)`	all	`s` reduced to the messages where predicate `p` returns true
`s.takeWhile(p)`	all	`s` up until predicate `p` returns false for a message
`s.dropWhile(p)`	all	`s` starting with the first message that predicate `p` returns false for
`s.map(f)`	all	`s`, with function `f` applied to each message
`s.mapSlice(f)`	Event	`map`-function on time slices. `f` maps from `UntimedStream` to `UntimedStream`

`s.scanl(f, u)`	all	(left) scan, *e.g.,* accumulating
		`Sync<1,3,5>.scanl(a->b->a+b, 0)`
		`== Sync<1,4,9>+`

`s.rougherTime(n)`	Event	`s` with every but each n'th `Tk` removed
`s.delay(n)`	Event, Topt	`s` with empty time slices prepended n times
`s.sizeEmpty-Timeslices()`	Event, Topt	amount of time slices without message in `s`
`s.untimed()`	Event, Topt, Sync	`s` converted to `UntimedStream`
`s.event()`	Topt, Sync	`s` converted to `EventStream`
`s.topt()`	Sync	`s` converted to `ToptStream`

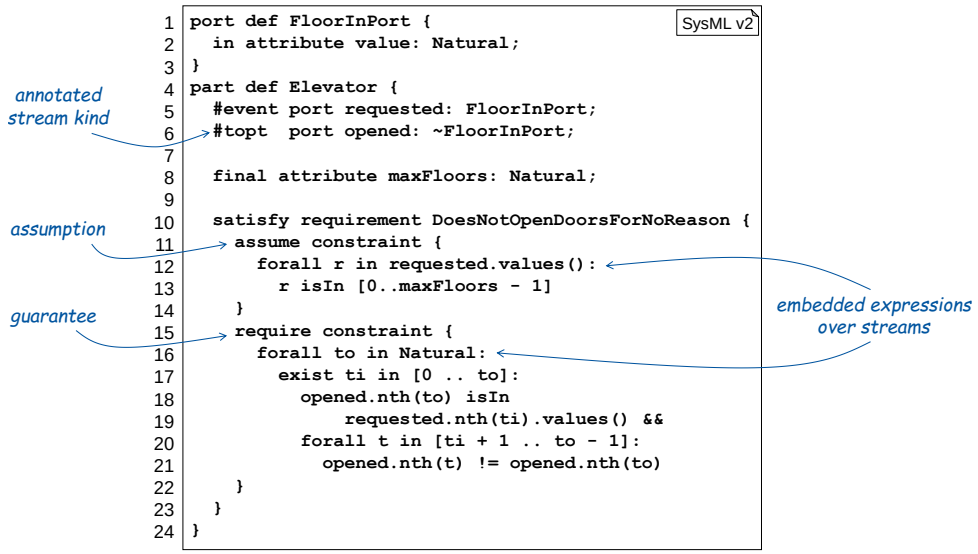
Table 1
A selected subset of stream operators. Most operators are applicable to streams of any kind, but some are specific to the timing information. `s` is a stream of any kind supported by the operator. `s2` is a stream of the same kind as `s`. `n` is a natural number (including 0).

Another set of operators help with time slice manipulation. The depicted `rougherTime`, `delay`, and `sizeEmptyTimeSlices` are examples of helper operator of common functionality. Their behavior could easily be recreated with the aforementioned operators, but inclusion of such raises the ease of use of the library.

Finally, conversion functions exist from more specific to less specific stream kinds. These simplify expressions over multiple streams of different kinds by forgoing some time information. Conversion from less specific to more specific kinds is unspecified, and thus not offered by default, as it requires some form of interpretation of the missing time information.

4. Assumption/Guarantee Formalism & Realization

We apply the stream library within an A/G formalism [21] and sketch an approach for applying such specifications in runtime verification. This formalism models system components through assumptions as constraints on the environment, while guarantees specify constraints that components must fulfill



Listing 1: Specification of a safety property for an elevator control unit: Open doors only on floors where call buttons were pressed or where the elevator was sent from the inside.

under those conditions.

4.1. Realization of an A/G Language

Black-box specifications are defined on the component interfaces, and their ports, respectively. Thus, a specification is defined over all ports but can refer only to specific ports. We choose to implement the A/G formalism derived from Focus [22], as it directly supports stream-processing functions, and it provides a mature methodology for specifying both safety and liveness properties. We embed the corresponding A/G language in the SysML v2 Architecture Description Language. This choice is motivated by practical considerations, as SysML is the de facto standard for systems engineering modeling.

Listing 1 depicts an A/G specification stating that an elevator should open its doors only on floors to which it has been called via a floor button press or sent via a cabin button press. This example is derived from the Elevator case study [23], which systematically analyzes the complexity of an elevator's software control system. Here, we focus on a single safety property to exemplify our approach to assuring functional correctness.

We design the system by declaring two ports: `requested` models the floor where the elevator is being sent to by event. This floor is expressed as a natural number. `opened` denotes the floor at which the elevator opens its doors. The elevator should not emit a signal via `opened` while moving. We model time according to the required specification granularity. Accordingly, the input port is an `EventStream`, as all button presses are events by nature. The port signaling on which floor the doors are open is an `ToptStream`, as it emits heartbeat signals only while present at a specific floor. Streams over input-output ports are inherently infinite because they must contain infinitely many timeslices to be well formed. Thus, there are no assumptions to be explicitly expressed about time. There is a need to specify that only valid floor numbers are allowed on all input ports, constrained by the parameter `maxFloors`. The specification is thus parameterized. The guarantee is split into two part-conditions: the first part specifies that the doors may open on floor a only if, in the past, there was a button press for floor a via `requested`, or a button press from inside the elevator pointing to floor a; the second part specifies that no doors are opened on any floor while the elevator is in motion.

The concrete implementation of our approach is a history-based A/G formalism with stream expressions tailored as extensions. As this example shows, stream expressions alone do not suffice for such a specification. First-order logic formulas are essential for expressing formal properties. Thus, an embedding of set and OCL-style expressions in accordance with [24] is applied.

No conversion functions are shown here because every operation is expressible on the same stream

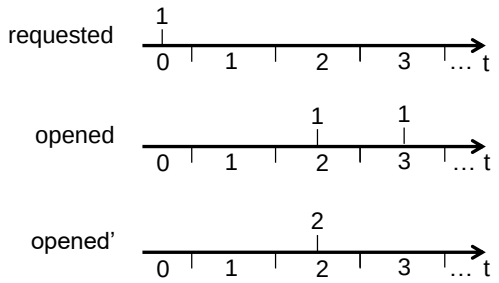


Figure 2: Input-output traces of the elevator control unit in relation to Listing 1. `requested` denotes the input trace of the component and `opened` the output trace of the component. Depending on the timeslices where the property is observed, the property is currently valid or currently invalid. `opened'` denotes the output trace where the property is definitely invalidated.

type. However, if any such expression is written in a type-independent way, conversion to a more appropriate type is encouraged.

4.2. Towards Runtime Analysis

We target monitoring and runtime verification of safety properties. For such properties to be monitorable and verifiable at runtime, specifications must either be monitorable by construction or be transformable into monitorable specifications [25]. Whether employing code-generation or implementing interpreters for implementing monitor synthesis from properties [26], it is imperative to be decidable at runtime if a property is monitorable or not. Additionally, a monitoring framework may attempt to monitor properties it does not find explicitly monitorable and produce an "unknown" verdict, which can then be interpreted by the user as appropriate.

Our underlying Focus formalism contains both executable and declarative expressions through stream operators and first-order logic, respectively, and specifications often combine both. This preserves expressiveness at a higher level of abstraction, but it also means that synthesis is required to derive monitors. Infinite-trace semantics [27] used in specifications has to be mapped to finite-trace semantics [25]. To derive verdicts, A/G specifications must therefore be interpreted using semantics similar to past-time LTL [28] in the Focus setting.

We revisit the example in Listing 1 and observe the input-output traces in Figure 2. If the property is observed by a monitor at time `t = 1`, the verdict is currently invalid because the elevator has not yet arrived at floor `1`. However, the trace can be extended to a valid one, for example at time `t = 2` on port `opened`. Thus, the verdict at `t = 1` is *currently invalid*. At time `t = 2`, we can likewise only conclude that the property is *currently valid*, since a suffix may still invalidate the property. For the output trace on port `opened'`, it is evident that the verdict changes from currently invalid at time `t = 1` to *definitely invalid* at time `t = 2`.

With this approach, we aim to derive monitors that provide weak and strong verdicts. A weak verdict states whether the property has been satisfied or violated up to the current time point, whereas a strong verdict states whether the property is definitively satisfied or violated [29]. A strong verdict can generally be given only if no suffix of the observed traces can change the expression's verdict. [30] defines a taxonomy of properties in relation to monitorability. Moreover, it was also proven in their work that all linear-time properties are monitorable if embedded in a truth domain with truth values `true`, `false`, `currently true` and `currently false`.

The example in Listing 1 highlights the need for a definition of safety over finite input-output traces in the stream-processing formalism employed in our approach. We aim to equip the A/G language with semantics for finite executions in accordance with [30]. A monitor can then provide users with *currently valid*, *currently invalid* and *always invalid from time t* verdicts, while still enabling meaningful, incremental monitoring. We further aim to place and map Focus liveness and safety concepts in the Safety-Progress classification of [30]. Based on this, we aim to transform Focus properties into equivalent linear-time properties in the Safety-Progress classification and derive monitorable obligation

properties.

5. Discussion

We chose to employ the stream library within an A/G formalism. However, stream-based specifications can also be expressed in other formalisms, such as equational, graphical, and relational ones [5]. We argue that the A/G style is particularly suitable for asserting properties in component-based architectural models, while relational styles are better suited to describing input-output interactions through predicates and equational styles are more implementation-oriented.

The A/G specification language defined in SysML v2 has a flexible structure in which multiple A/G pairs can be defined per component. For single components, this can be interpreted as assigning multiple asserted properties to one component. In a decomposed system, a component specification is valid only if all properties defined by A/G pairs are valid, which corresponds to a conjunction of the associated properties. At present, we target monitor derivation for complete specifications; modularity concerns are left for future work.

Timed streams such as `EventStreams` are, by default, infinite when observed at component interfaces through ports, because they contain infinitely many time slices. This raises the question of whether assumptions about (in)finiteness of input/output streams should be made explicit in the model or remain implicit in realizations and be disallowed at the specification level. Furthermore, although assumptions over complete input/output stream lengths are not allowed, assumptions about the lengths of individual time slices can still be meaningful for modeling the environment.

Another concern is that assumptions in classic Focus A/G can only refer to input ports and parameters. This separation supports well-formedness based on syntactic rules, but it does not guarantee that assumptions involving both input and output cannot be encoded in the guarantee part. Actually, everything could be incorporated in the guarantee as implications and the semantics would remain unchanged.

Expressing time at ports has advantages, as it enables different stream kinds to be used directly as interface types. However, more common specification styles associate timing with the specification as a whole and use only one timing per specification [5]. One reason for this design is that a component interface does not necessarily reveal how a component internally processes time. A more refined interface design could expose only whether behavior is timed or untimed, while defining the exact timing model in the behavior itself; however, this may introduce additional syntactic clutter.

Without restricting specification expressiveness, it is possible to define properties that are impractical for runtime analysis from a performance perspective and therefore require offline runtime verification. The impact of synthesized monitors on the performance of the monitored system is another important topic that remains for future evaluation.

Finally, we focused on monitorability because it strongly influences the overall runtime verification design and omitted the monitoring setup, architecture and workflow of our approach. We chose to delegate this to future work.

6. Conclusion & Future Work

We presented a black-box approach for specifying the functional behavior of components by embedding an A/G specification notation over component input–output histories into SysML v2, enabling engineers to formally specify behavior in terms of assumptions and guarantees. This notation supports describing how a component should behave while deliberately omitting how this behavior is achieved, *i.e.,* allowing specifications to remain intentionally underspecified. Such underspecification enables engineers to refine models incrementally, gradually adding detail and tightening constraints as the design matures.

Beyond the specification language itself, we sketched how a computable subset of specifications can be transformed into executable monitors for runtime analysis. Our approach is indicative of monitor synthesis for certain types of decidable, functional behavioral constraint violations.

Future work entails extending transformation capabilities; We only sketched a simple transformation algorithm to derive monitors. However, we should be able to extend our approach to encompass a broader class of computable specifications. The soundness of said transformation must then be proven to ensure the correctness of the monitors without proving soundness for each generated monitor individually. We are also interested in integration into toolchains for the automated generation of runtime monitors, *e.g.*, in the context of digital twins, to increase usability for industry-focused projects.

Declaration on Generative AI

During the preparation of this work, the authors used Prism, GPT-5.3/5.4 and Gemini 3.1 in order to: Grammar and spelling check, improve writing style. After using these tools/services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] A. W. Wymore, Model-based systems engineering, CRC press, 2018.

[2] B. Rumpe, A. Wortmann, Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures, in: Lohstroh, Marten and Derler, Patricia Sirjani, Marjan (Ed.), Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday, LNCS 10760, Springer, 2018, pp. 383–406.

[3] M. Bajaj, S. Friedenthal, E. Seidewitz, Systems modeling language (sysml v2) support for digital engineering, Insight 25 (2022) 19–24.

[4] E. Guerra, J. de Lara, On the quest for flexible modelling, in: Proceedings of the 21th ACM/IEEE International conference on model driven engineering languages and systems, 2018, pp. 23–33.

[5] M. Broy, K. Stølen, Specification and development of interactive systems: focus on streams, interfaces, and refinement, Springer Science & Business Media, 2012.

[6] J. O. Ringert, B. Rumpe, A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing, International Journal of Software and Informatics (2011).

[7] M. Broy, Theory and methodology of assumption/commitment based system interface specification and architectural contracts, Formal Methods in System Design 52 (2018) 33–87.

[8] V. Aravantinos, S. Voss, S. Teufl, F. Hölzl, B. Schätz, Autofocus 3: Tooling concepts for seamless, model-based development of embedded systems., ACES-MB&WUCOR@ MoDELS 1508 (2015) 19–26.

[9] N. Alzahrani, M. Spichkova, J. O. Blech, From temporal models to property-based testing, arXiv preprint arXiv:1705.10032 (2017).

[10] S. Kanav, V. Aravantinos, Modular transformation from af3 to nuxmv., in: MODELS (Satellite Events), 2017, pp. 300–306.

[11] H. Kausch, M. Pfeiffer, D. Raco, B. Rumpe, A. Schweiger, Model-driven Development for Functional Correctness of Avionics Systems: A Verification Framework for SysML Specifications, Council of European Aerospace Societies Aeronautical Journal (CEAS 16 (2025) 33–48.

[12] H. Kausch, M. Pfeiffer, D. Raco, B. Rumpe, A. Schweiger, Enhancing System Model Quality: Evaluation of the Systems Modeling Language (SysML)-Driven Approach in Avionics, Journal of Aerospace Information Systems (JAIS) 22 (2025) 367–378.

[13] H. Kausch, M. Pfeiffer, D. Raco, B. Rumpe, S. Stüber, L. Wollenhaupt, Towards an Isabelle Theory for Distributed, Interactive, Real-Time Systems Volume 2, Aachener Informatik-Berichte, Software Engineering, Band 58, Shaker Verlag, 2025.

[14] W. Ahrendt, J. M. Chimento, G. J. Pace, G. Schneider, A specification language for static and runtime verification of data and control properties, in: International Symposium on Formal Methods, Springer, 2015, pp. 108–125.

[15] C. Colombo, G. J. Pace, Runtime verification using larva (2017).

[16] F.-M. Adolf, P. Faymonville, B. Finkbeiner, S. Schirmer, C. Torens, Stream runtime monitoring on uas, in: International Conference on Runtime Verification, Springer, 2017, pp. 33–49.
[17] P. Faymonville, B. Finkbeiner, M. Schledjewski, M. Schwenger, M. Stenger, L. Tentrup, H. Torfah, Streamlab: stream-based monitoring of cyber-physical systems, in: International Conference on Computer Aided Verification, Springer, 2019, pp. 421–431.
[18] J. Baumeister, B. Finkbeiner, S. Schirmer, M. Schwenger, C. Torens, Rtlola cleared for take-off: monitoring autonomous aircraft, in: International Conference on Computer Aided Verification, Springer, 2020, pp. 28–39.
[19] F. Gorostiaga, C. Sánchez, Runtime verification of real-time event streams using the tool hstriver, Formal Methods in System Design 61 (2022) 3–34.
[20] S. Marlow, et al., Haskell 2010 language report, 2010.
[21] B. Meyer, Applying'design by contract', Computer 25 (2002) 40–51.
[22] M. Broy, Towards a theory of architectural contracts:-schemes and patterns of assumption/promise based system specification, in: Software and Systems Safety, IOS Press, 2011, pp. 33–87.
[23] F. Strobl, A. Wisspeintner, Specification of an elevator control system–an autofocus case study (1999).
[24] B. Rumpe, Modeling with UML: Language, Concepts, Methods, Springer International, 2016.
[25] E. Bartocci, Y. Falcone, A. Francalanza, G. Reger, Introduction to runtime verification, in: Lectures on Runtime Verification: Introductory and Advanced Topics, Springer, 2018, pp. 1–33.
[26] H. Barringer, K. Havelund, Internal versus external dsls for trace analysis, in: International Conference on Runtime Verification, Springer, 2011, pp. 1–3.
[27] M. Broy, et al., A tutorial on runtime verification, Engineering dependable software systems 34 (2013) 141.
[28] K. Havelund, D. Peled, An extension of first-order ltl with rules with application to runtime verification, International Journal on Software Tools for Technology Transfer 23 (2021) 547–563.
[29] H. Barringer, D. Rydeheard, K. Havelund, Rule systems for run-time monitoring: from eagle to ruler, Journal of Logic and Computation 20 (2008) 675–706.
[30] Y. Falcone, J.-C. Fernandez, L. Mounier, What can you verify and enforce at runtime?, International Journal on Software Tools for Technology Transfer 14 (2012) 349–382.