SE | RWTH AACHEN UNIVERSITY

Hendrik Kausch, Mathias Pfeiffer, Deni Raco,
Bernhard Rumpe, Sebastian Stüber,
Lucas Wollenhaupt

Towards an Isabelle Theory for Distributed, Interactive, Real-Time Systems Volume 2

Aachener Informatik-Berichte,
Software Engineering, Band 58
Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Abstract

In many applications, the behavior of a component depends on the time when messages are received. To model these in embedded systems, capabilities to specify time are required. This includes the capability to react to the absence of input. In this report, we present an encoding of FOCUS [BS01b, Bro14, Bro23a, Bro24] in the theorem prover Isabelle. This implementation extends our previous formalization of untimed streams [BKR+20]. Similar to the untimed version, concepts such as timed stream bundles, timed stream processing functions, and corresponding functions and theorems are presented. The principle idea is to conceptualize the observable flow of messages over a channel as a stream and the behavior of a component as a stream processing function. A component's specification is then given by a set of stream processing functions, allowing for the modeling of underspecified behavior. Refinement and composition of components are natural operations in this theory and are compatible. This is a great advantage when modular reuse, evolutionary optimization, or incremental development are required to develop highly reliable systems that must be certifiable or even verifiable. The theories are evaluated by proving the properties of a time-sensitive case study.

[KPR+25c] 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,
DOI 10.2370/9783819100628, Shaker Verlag, Jun. 2025.

Contents

Chapter 1

Introduction to the Timed Stream Formalization

Software is getting more complex for various reasons, including increased user demands for more functionality, integration with multiple systems and devices, technological advancements, and higher security. As software systems evolve and become more sophisticated, developers face the challenge of creating easy-to-use, secure, and reliable applications while incorporating complex features and functions. This complexity results in an increase in the size and intricacy of the software systems, making designing, building, and maintaining them more challenging [Bro06a]. Model-based system development can help with dealing with this complexity. By modeling a software system as a distributed system comprising various components communicating with each other, we can analyze the system and reason about properties such as security and reliability. In many applications, the behavior of a component depends on the time when messages are received (e.g., in the case study we considered in this work, see Chapter 6). For instance, the timely activation of an airbag can be seen as a (critical) functional property. In untimed streams [BKR$^+$20], an implicit concept of time was present as a total order on the set of events. The order of incoming messages was modeled, but not the distance in time between arrivals. It is important to note that timed streams enable a stronger expressiveness compared to untimed streams. Timeouts, a very common construct in real-time embedded systems, can only be modeled by timed streams (through the capability to react to the absence of input). Furthermore, such properties become quantifiable over time. We quantify the time of arrival of events or execution of actions. We introduce a discrete model (and no continuous dense model) [BS01b, Rum96]. Timed modeling uses, apart from an alphabet `M`, also a special message $\surd$, called `Tick`, in order to model the passage of time. So, the occurrence of a `Tick` in a stream models the incrementing of time by one time unit. Additionally, we assume equality of length of all time units to avoid Zeno effects. The n-th `Tick` of a stream models then the end of the n-th time unit. The only known information about the incoming messages between the n-th and the n+1-th `Tick` is their order and occurrence in the time interval [n, n+1]. Similarly to the untimed streams [BKR$^+$20], one can define timed versions of stream bundles and stream processing functions. The untimed theory developed in [BKR$^+$20], including composi-

tion and refinement techniques, can thus be seamlessly transferred. The realizability of timed stream processing functions is described through strong and weak causality (see [Rum96]), also known as "pulse-driveness", and it replaces the previously required properties of monotonicity and continuity. The model of time in this paper is discrete and similar to the one proposed in [Ost89]. Section 3.3 discusses further alternatives. Models of dense time where infinite points between two finite points are used are presented in [Lyn89]. This is based on the concept of the `dense time domain` [Bro12]. In [BS01b], timed streams are defined as infinite observations, where a timed stream has infinitely many `Ticks`. Several discrete and dense variants of streams are presented in [RR11]. Dense models of time are usually attractive to specify physical hardware phenomena, while our discrete time is rather suited for software aspects. The stream implementation in Isabelle is based on the "Higher-Order Logic of Computable Functions" (HOLCF) [Reg94] extension. Our previous work [GR06] introduced a datatype for general streams and defined functions over these streams. The current definition of streams is an optimization of that definition based on the more recent HOLCF'11 [Huf12]. Another approach of a FOCUS formalization in Isabelle is presented in [Spi07]. It is based on Isabelle/HOL instead of Isabelle/HOLCF and uses the `list` data type. A detailed discussion of alternatives can be found in Section 3.3. Subsets of the work presented in this book have been applied to case studies from domains such as automotive and aerospace. In [KRRS19], a time-synchronous model was used and evaluated in an automotive case study. The Isabelle implementation was connected with the frontend modeling language MontiArc [HRR12]. MontiArc and a time-synchronous model were used again in [KPRR20b] for proving certain properties of a deterministic pilot flying system from a NASA case study [CM14]. A nondeterministic variant of the pilot flying system was refined step-wise and proven correct in [KPRR20a] using MontiArc and its time-synchronous model. Next, the prominent language SysML was used as the frontend to specify the pilot flying system using a time-synchronous model [KPRR21], which is known to be more commonly used for hardware modeling. An event-driven model, known to be used in distributed software applications, was applied in [KMP+21], where the pilot flying system was modeled again using the SysML language in MontiCore [HKR21, HR17, GKR+06]. Further communication and architectural pattern for distributed systems are discussed in [BKP+25]. Using the event-driven model, another case study from the aerospace domain, a data link uplink feed, was modeled with SysML and verified by a mapping to Isabelle [KPR+22]. Finally, in [KPR+23], the data link uplink feed case study was also modeled in MontiArc, and the fundament of event-driven processing using FOCUS was presented.

1.1 Goals and Results

The key contributions of this work are:

- An optimized Isabelle realization of timed streams.
- A library of useful functions on timed streams, bundles, stream processing functions, and specifications.
- A unique implementation of the causality in Isabelle and its corresponding proofs.
- A formalization of realizability constraints for timed modeling of components and component networks with potential feedback loops.

This technical report is structured as follows. We formally introduce the concepts and Isabelle-encoding of timed streams in Chapter 3. Several variants of the timed stream, including the time-synchronous stream, are formally introduced in the same chapter. In Chapter 4, we then introduce the formalization and the implementation of timed stream bundle in Isabelle. Based on the timed stream bundle, the formalization of stream-processing functions and stream processing specifications, as well as the composition in the timed case, are discussed in Chapter 5. Finally, we employ the above in a timed case study in Chapter 6.

Chapter 2

Preliminaries

This chapter introduces preliminaries to the FOCUS formalism [BS01b] and the theorem prover Isabelle [NPW02]. It also summarizes related formalisms and theorem provers.

2.1 Focus

In the mathematical framework FOCUS [BS01b, BR07, RR11, Bro23b, Bro23a, Bro24] distributed interactive systems are modeled as components communicating via directed channels with each other. An example component with two input and one output channels is depicted in Fig. 2.1. Sequences of messages depicting the communication history of channels are called streams [RR11]. The passing of time in a communications history can be depicted by adding additional symbols, i.e., a $\surd$ denoting the end of a time slice.

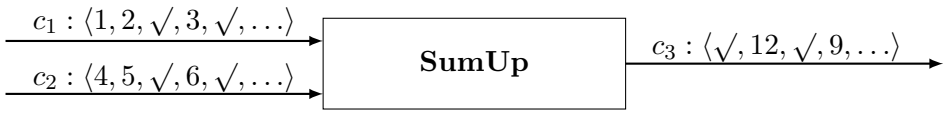

Figure 2.1: Component sums up the inputs from both input streams between consecutive $\surd$ pairs.

Building upon formal specifications in FOCUS, the considered systems semantics [HR04, Rum98], even infinite networks of components [Bro87], can be defined and analyzed, e.g., the correctness can be proven mathematically. Furthermore, the formal basis of FOCUS allows underspecification. Thus, the formalism can be used even when the system is not fully developed, e.g., to verify strongly underspecified or non-deterministic systems [Bro86], early in the design phase. In FOCUS, components can be specified through relations between inputs and output, state-based, or by composing other components. State-based specifications, their behavior, and their refinement properties in FOCUS are detailed in [Rum96, Bro23a]. Through behavior *refinement*, i.e., removing possible behavior, further development of a verified system is facilitated without proving

the correctness again [BKRW17, BR23]. Each system component can be decomposed into a subsystem. This allows abstract description of complex systems, while providing more concrete details over multiple hierarchy levels [MRR13, BR07, Bro10a, RW18] as depicted in Fig. 2.2. Especially important is the fact that in FOCUS, behavior refinement of a subcomponent implies behavior refinement of the composed system. Using this, complex and hierarchically decomposed systems can be verified following the "divide and conquer" principle, breaking the proof down into simpler proofs over the components. This also works over multiple hierarchical levels, i.e., the refinement relation is transitive over architectural decomposition. Additionally, a formal correct architectural refactoring of a system's structure is feasible by applying suitable refactoring steps [PR01] leading to a simplified step-wise refinement of an architecture [PR97, PR99, PR03, FKR+25]. In order to allow the specification and verification of time and time-dependent system,

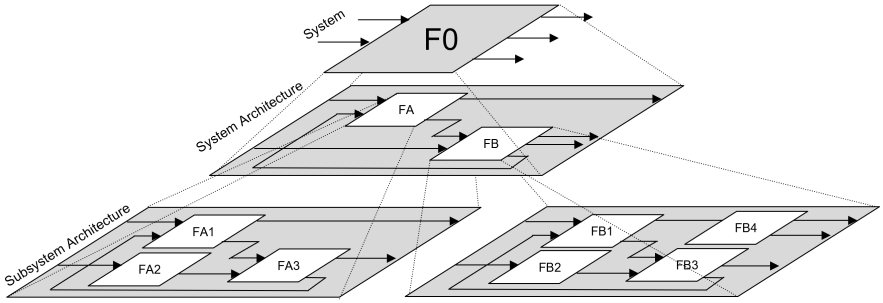

Figure 2.2: Hierarchical decomposition of a system. From lecture "Innovations in Software Engineering" by Prof. Rumpe, 2019

causality is an essential factor for realizability [Bro23b]. An in-depth presentation of the FOCUS-calculus is in [Bro24]. It explains how to enables proofs of system properties through interface assertions and architectures composed of subsystems. It is shown to be sound and relatively complete, handling specifications with timing properties and allowing proofs of real-time and causal properties. The calculus supports two types of logical deductions: classical predicate logic and stronger causality and full realizability deductions, assuming systems are implemented by generalized Moore machines. This modular approach aligns with engineering principles like encapsulation and information hiding. Verification of individual and composed system properties is possible.

In [BS01b], three distinct temporal abstractions are introduced for specifying distributed systems. The untimed abstraction has no temporal considerations, i.e., nothing about time is specified. The time-synchronous abstraction enforces strict synchronization by allowing exactly one message to be transmitted per discrete time frame. In contrast, the timed abstraction allows a bounded, finite number of messages within each time frame. Of the three, the timed abstraction is the most general and expressive, capable of representing both untimed and time-synchronous specifications.

Each abstraction level is suited to different system requirements and use cases, depending on the required level of temporal precision. Beside behavior refinement, additional refinement kinds, including communication history refinement, and interface refinement are defined in [Bro93b] and provide refinement options for a methodological development of distributed systems.

Slightly extending FOCUS by partial communication histories, where the respective channel can be active or inactive for specific time-slices allows the specification of discrete *dynamic* distributed systems [Bro14, HRR98]. Thus, components can dynamically change the systems infrastructure by either using or not using communication channels for a specific time-frame. This may lead to underspecified behavior when different components use the same channel in the same time-slice for transmitting messages, e.g., for some parallel compositions. The resulting dynamic architecture is similar to a publish/subscribe architecture, where in each time-slot components can either publish or subscribe to a communication channel. Dynamic interface assertions and state-machines define the internal behavior of components in a dynamic system. For dynamic state-machines, a Mealy machine [Mea55] with infinite state-space is extended by allowing to switch active and inactive channels and changing the state-space structure for each time-slice. Using interface assertions as a descriptive logical specification instead, the standard assertions from [BS01b] are extended to allow checking channels activeness status and allowing transmitting channels as messages, similar to the π-calculus introduced Section 2.1.1.

A way to formally specify service oriented system in FOCUS is presented in [BKM07, Bro10b]. Services differ from components in that services are partial, i.e., services' behavior is only defined for a subset of input histories. Three techniques are used to specify services: assumption/commitment specifications, state machines, and composition of sub-services. Assumption/commitment specifications outline input conditions (assumptions) for correct service behavior and guarantee outputs (commitments) when assumptions hold. State transition diagrams [GKRB96, RK96, PR94] represent services through states and transitions, where the responses is based on the current state and input. Composition of sub-services allows complex services to be constructed hierarchically from smaller sub-services, with each fulfilling part of the overall behavior. This decomposition supports a structured approach to handling the complexity of large, service-oriented systems [HLMSN$^+$15, KOB$^+$17]. The methodology promotes a service-oriented view on architectures, emphasizing a hierarchical structure. By breaking down systems into individual services and components, the approach facilitates modular verification and scalable design. Combined, these techniques provide a powerful theoretical foundation for specifying and verifying the behavior of complex service oriented systems.

In [BS24], the FOCUS formalism provides the formal underpinning for describing layered architectures. Within this model, lower layers provide functional capabilities (services) to upper layers. Lower layers without any service dependencies act as platforms for the upper layers. Each layer has a required interface representing the needed ser-

vices and a provided interface representing the provided services. A layer hides internal communication between its services from other layers. Composing layers together results again in a layer. Key properties of FOCUS are retained, e.g., the refinement stays compositional [Bro97a] over layers. Furthermore, from strong causality requirements, it can be ensured that as long as required servies of a layer are guaranteed, the provided services will be delivered. In conclusion, an abstract and high-level specification of layered architectures with clear semantics but without considering implementation-oriented details is defined.

The mathematical underpinning of FOCUS can be extended and tailored to specific needs, i.e., for defining discrete, continuous, or hybrid systems. In comparison to strictly discrete streams of data described in [BS01b], continuous streams of data are additionally modeled in [Bro93a, Bro97b, Bro12]. Here, streams are represented by mapping elements of a time domain to messages. A time domain is a subset of $\mathbb{R}_+$. It is possible to have discrete streams (e.g., with time domain $\mathbb{N}_0$), or dense time (e.g., with time domain $\mathbb{R}_+$). Furthermore, refinement relations between abstract, discrete and concrete continuous components allow easy behavior-specification for hybrid systems by discretizing a continuous stream to its important events, e.g., the filling quantity of a tank is below 10%. The basic datatypes of FOCUS are streams, stream bundles (SBs), and stream-processing functions (SPFs). The communication history of channels is represented by sequences of messages called streams. The concept of streams and timed streams is introduced in Chapter 3. Components are represented as SPFs mapping input streams to output streams. The SB datatype bundles streams of component interfaces together and is introduced in Chapter 4. Each SPF characterizes one possible behavior of a component. Deterministic and non-deterministic components are introduced in Chapter 5. FOCUS can be used as a general semantics-giving formal foundation for software and systems modeling and formal analyses [BHP+98, RKB95, BFH+10]. In [BŞ01a] the formal semantics of data flow networks represented by flownomial expressions are defined using SPFs and stream processing specifications (SPSs). Verification tools employing the FOCUS methodology include the Isabelle-based FOCUS stream implementation from [Spi07] and the AutoFOCUS tool [BHS99]. The Isabelle stream implementation merges four stream types (timed/untimed and finite/infinite), which complicates induction on general streams since it would require separate inductions for each subtype. Despite this complexity, [Spi07] successfully defined system components and verified requirements within Isabelle/FOCUS. AutoFOCUS, on the other hand, uses Moore machines to define component interfaces, refinement, and behavior, with a primary emphasis on the graphical representation of systems and components. Basic validation within AutoFOCUS is achieved via context conditions, ensuring properties like type correctness. Verification of refinement relations uses logical implications defined on specification predicates. AutoFOCUS includes a model-to-code transformation, with a verified C0-code generator [HST10] that translates models into executable Isabelle theories. The application of AutoFOCUS in the automotive industry is described in [Bro06b] as part of a transformation

for the software and systems engineering for embedded systems in cars. Using FOCUS to define the semantics of a language was demonstrated in the telecommunication domain for the Specification and Description Language (SDL) [Bro91].
Furthermore, FOCUS is the formal foundation for the SPES methodology and SPESML[1] [PHDB16]. SPESML was developed with various industrial partners from the automotive, avionic, and healthcare domain. The goal is to handle the growing complexity of distributed cyber-phisical systems [BCG12, Bro13, BS14] with model-based systems engineering and tool support. Building upon SPES and FOCUS [BKPS07], a verification framework for avionic systems and a general methodology for modeling and verifying systems was developed and applied to a Data-Link Uplink Feed case-study [KPR+25b]. The benefits of the methodology regarding model quality is analyzed in [KPR+24, KPR+25a]. Furthermore, [KKN+24] describes how the implementation in combination with a transformation for models to Isabelle encodings can verify AI generated SysML models.

2.1.1 Alternative Formalisms

Besides FOCUS, other formalisms for formal specification and verification of distributed systems exist. Some domains are especially interested in specific kinds of requirements, e.g., probabilistic requirements [Bir07, ALRL04] for cryptographic systems. The general formalism FOCUS is extendable for such cases when necessary, an extension for probabilistic systems is shown in [Neu12]. We now introduce the main alternatives. While all alternatives offer excellent specifications and some also have great verification properties, FOCUS has the same capabilities while also providing important properties for iterative system development, i.e., the refinement of a subcomponent refines the whole system.

Calculus of Communicating Systems (CCS)

The Calculus of Communicating Systems (CCS) was first introduced by Robin Milner in his book "A Calculus of Communicating Systems" [Mil80] in the year 1980. It is inspired by processes, often referred to as agents, that communicate via a medium. Each agent has ports, called labels, through which communication with the environment is possible. A label has two complementary parts, one of which is an input label and the other one is an output label. If l is an arbitrary input label, $\bar{l}$ is the output label corresponding to it. CSS provides the following important operators:

- Action-prefixing operator: For an agent P and action α, $\alpha.P$ is the agent that first executes α and then behaves like the agent P.
- Sum operator: For two agents P and Q, $P + Q$ is the agent, which behaves like agent P or agent Q.

[1] `https://spesml.github.io`. Accessed 22.05.2025

- Composition operator: For two agents P and Q, $P|Q$ defines the agent, which executes both P and Q simultaneously. This operator enables the processes P and Q to communicate with each other via shared labels.
- Restriction operator: For the set of labels $L = l1, \ldots, ln$ and an agent P, $P\backslash L$ describes the agent that behaves like P, but all labels in L (including their counterparts) are not usable for communication to the environment.
- Conditional operator: For agents P and Q and expression e, the conditional operator `if` e `then` P `else` Q, describes an agent that behaves like agent P if e evaluates to true and like Q if e evaluates to false.
- Relabelling operator: For an agent P and a function f, which renames labels to arbitrary new labels, $P[f]$ is the agent, which has all labels renamed according to f.

With those operators, it is possible to specify complex models with multiple processes. The calculus is further extended to the π-calculus [Mil99], where components can communicate names of communication channels to allow dynamic reconfiguration of distributed systems.

Communicating Sequential Processes (CSP)

Communicating Sequential Processes (CSP) is a formal language introduced by C. A. R. Hoare in 1978 [Hoa78]. However, CSP has evolved substantially since then. The most used version of today is explained in A. W. Roscoe's book "The Theory and Practice of Concurrency" [Ros97]. Hoare described that processes or programs are fundamentally based on the primitives of input and output [Hoa78]. Thus, in CSP, processes communicate via channels with each other (similar to CCS), and the behavior of parallel executions is specified by a composition operator. Another important concept is nondeterminism, which can be modeled using the non-deterministic choice operator. CSP is used as the theoretical foundation for multiple refinement and model checking tools, e.g., ProB [LF08] and Process Analysis Toolkit [SLD08].

Temporal Logic of Actions (TLA)

In his 1983 paper "The Temporal Logic of Actions" [Lam94], Leslie Lamport formally specifies the Temporal Logic of Actions (TLA). According to Lamport, using mathematical logic to specify the behaviors of complex concurrent systems is easier than writing code. TLA is a logic where safety, liveness, or fairness expressions can be formulated to define or reason about a specification, which might be a system or properties. A TLA formula is defined using state functions and special operators $\prime$, $\Box$, $\exists$. [AL95] State functions assign values to states using boolean known operators $\vee$, $\wedge$, $\Rightarrow$, and $=$. If

a state function is boolean-valued, it is a state predicate. Using variables and primed $'$ variables in a state predicate is a so-called action. An action defines possible state changes in the system. One way to reason and specify over time is to use the temporal logic operators like $\Box$ and $\diamond$. If a formula f should hold at all times, $\Box f$ expresses that if it should hold eventually, write $\diamond f$. A behavior, then, is an infinite sequence of states. A system behavior consists of all possible behaviors of the system. A behavior is possible if and only if an action exists from the old state to the new one for each state change. Furthermore, the behavior must meet the requirements of the specification, and its initial state has to fulfill its initial state predicate. So far, specifications can use time only in a limited form. One could argue that each "step" of a system occurs at some time, and previous steps occur earlier. Thus a system can "count" time in regards to its own steps. However, these system steps may take longer or shorter; they are not comparable to real-time progress, and the system can never really react to the passage of time in the sense of physical time. The general specification approach of TLA was extended to make real-time specifications possible. The main idea to be able to reason about real-time and its progress in discrete systems is to extend the TLA specification by a special variable known as *now*. [AL94, Lam99] The specifications of systems work with discrete steps, as explained above. The discrete "steps" of TLA specifications define the observable steps of a real physical system and can be seen as "snapshots" of a system's state. Since such snapshots define a discrete sequence of a system's states, the *now* variable used for defining the system is observed as a discrete sequence as well. Thus, even though real-time advances continuously, specifying and reasoning about a system in TLA only needs discrete time and system steps. Still, some general assumptions for the *now* variable in any specification must hold to guarantee time progression. A predicate ensures that the *now* variable will eventually go up and thus not infinitely much time passes between two discrete time steps. Generally, time advancement and system steps are divided such that they do not happen simultaneously. The extension TLA$^+$ [Lam99] introduces additional specification operators to aid user specification. The Temporal Logic of Actions Proof System (TLAPS) [CDL$^+$12] allows behavior verification based on model checker or, for complex cases, a theorem prover.

2.2 Domain Theory underlying Focus

To formalize the semantics of Focus, one requires a mathematical theory in which the semantics can be described. For Focus this is domain theory [SK95, Chapter 11]. A domain is a set of mathematical objects. Employing domain theory, one can formalize the denotational semantics of programming languages with regards to recursive datatypes and recursive definitions [AJ95]. In the following, some definitions are introduced, which are particularly important for the remaining exposition. These definitions are based on [SK95]. Furthermore, Kleene's fixed-point theorem [Kle52] is introduced.

Orders

First, we define the fundamental structural properties of sets using a binary relation called order.

Definition 2.1 (Partial order)**.** A partial order (po) over a set D is a binary relation $\sqsubseteq$ over D that has the following properties:

- reflexivity: $\forall x \in D.\ x \sqsubseteq x$
- transitivity: $\forall x, y, z \in D.\ x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$
- antisymmetry: $\forall x, y \in D.\ x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$

The subset relation as an order on the powerset of a set $\mathbb{S}$, i.e., $(\mathbb{P}(\mathbb{S}), \subseteq)$, is a po. In general, a po is not total, i.e., there may be incomparable elements. Thus, we define total orders as follows.

Definition 2.2 (Total order)**.** A total order is a po $(D, \sqsubseteq)$ with totality, that is, $x \sqsubseteq y$ or $y \sqsubseteq x$ for all $x, y \in D$.

For instance, the less-than relation on the integers, i.e., $(\mathbb{Z}, \leq)$, is a total order. Next, we define certain special elements of subsets for a given po.

Definition 2.3 (Minimal element)**.** Let $(D, \sqsubseteq)$ be a po and $X \subseteq D$. An element $m \in X$ is called minimal if for all $x \in X$ we have

$$x \sqsubseteq m \implies x = m.$$

In $(\mathbb{Z}, \leq)$ there is no minimal element while in $(\mathbb{N}, \leq)$ the minimal element is 0. However, in general, a minimal element is not unique. In $(\mathbb{P}(\{0, 1\}) \setminus \{\emptyset\}, \subseteq)$ for example, both $\{0\}$ and $\{1\}$ are minimal. We thus define the least element as the unique minimal element.

Definition 2.4 (Least element)**.** Let $(D, \sqsubseteq)$ be a po and $X \subseteq D$. An element $l \in X$ is called least element of X if $l \sqsubseteq x$ for all $x \in X$.

For example, 0 is the unique minimal element and thus the least element in $(\mathbb{N}, \leq)$.

Definition 2.5 (Upper bound)**.** Let $(D, \sqsubseteq)$ be a po and $X \subseteq D$. An element $b \in D$ is called upper bound of X if $x \sqsubseteq b$ for all $x \in X$.

As with minimal elements, upper bounds are not unique in general. All finite subsets of $\mathbb{N}$ have infinitely many upper bounds in $(\mathbb{N}, \leq)$, namely the maximal element in the subset and every number greater than that. However, an infinite subset such as $\mathbb{N}$ itself has no upper bound. If an upper bound exists, we are interested in a particular one called the least upper bound. It has the benefit of being unique.

Definition 2.6 (Least upper bound)**.** Let $(D, \sqsubseteq)$ be a po and $X \subseteq D$. The least upper bound (lub) of X, which we denote by $\bigsqcup X$, is an upper bound of X such that for all upper bounds $b \in D$ of X we have

$$\bigsqcup X \sqsubseteq b.$$

The lub is thus the least element in the set of upper bounds and hence unique. Next, we define a special kind of subset.

Definition 2.7 (Ascending Chain)**.** Let $\sqsubseteq$ be a po on D. A countable non-empty subset $\{c_1, c_2, c_3, \dots\}$ of D is called an ascending chain if

$$c_1 \sqsubseteq c_2 \sqsubseteq c_3 \sqsubseteq \cdots .$$

Note that an ascending chain is a totally ordered subset and has a minimal element, which is also the least element in that subset due to totality. Generally, the lub of chains in a po does not exist. For instance, $\mathbb{N}$ is an ascending chain in $(\mathbb{N}, \leq)$ and has no lub. However, if we define $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ and $n \leq \infty$ for all $n \in \mathbb{N}$ then $\mathbb{N}$ is an ascending chain in $(\mathbb{N}_\infty, \leq)$ and has a lub, namely $\bigsqcup \mathbb{N} = \infty$.

Definition 2.8 (Complete partial order)**.** A complete partial order (cpo) is a po $(D, \sqsubseteq)$ with the additional property: that for each ascending chain $C \subseteq D$ the lub $\bigsqcup C \in D$ exists.

A simple case of a po is $(D, =)$ for any set D and equality as the relation. This construction is referred to as discrete partial order. The discrete partial order over any set D is even complete. A cpo does not necessarily contain a least element. For instance, every discrete partial order with at least two elements does not have a least element since each element is only comparable to itself.

Definition 2.9 (Pointed complete partial order)**.** A pointed complete partial order (pcpo) is a cpo $(D, \sqsubseteq)$ which has a least element in D called bottom denoted by $\bot \in D$.

Every cpo can be lifted to a pcpo by introducing a new bottom element. As a result, combined with the discrete partial order, any set can be lifted to a pcpo.

Functions

Next, we consider functions mapping between two pos and define important properties.

Definition 2.10 (Monotonic functions)**.** A function $f \colon D_1 \to D_2$ with pos $(D_1, \sqsubseteq_{D_1})$ and $(D_2, \sqsubseteq_{D_2})$ is monotonic if

$$x \sqsubseteq_{D_1} y \implies f(x) \sqsubseteq_{D_2} f(y)$$

for all $x, y \in D_1$.

Definition 2.11 (Continuous functions)**.** A function $f\colon D_1 \to D_2$ with cpos $(D_1, \sqsubseteq_{D_1})$ and $(D_2, \sqsubseteq_{D_2})$ is continuous if and only if

$$f\left(\bigsqcup C\right) = \bigsqcup f(C)$$

for all ascending chains $C \subseteq D_1$.

$f(C)$ is shorthand for the element-wise application of f, i.e. $f(C) = \{f(c) \mid c \in C\}$. Note that continuity implies monotonicity. However, the converse does not hold; monotonicity does not imply continuity. Next, we consider functions mapping a set to itself and define the notion of a fixed point.

Definition 2.12 (Fixed point)**.** An element $d \in D$ for some set D is a fixed point of the function $f\colon D \to D$ if $f(d) = d$.

Some functions have no fixed point such as $f\colon \mathbb{N} \to \mathbb{N}, x \mapsto x + 1$. Others may have a unique fixed point such as $g\colon \mathbb{N} \to \mathbb{N}, x \mapsto 1$. And still others have infinitely many fixed points, e.g., $h\colon \mathbb{N} \to \mathbb{N}, x \mapsto x$. If a fixed point exists we are often interested in a unique one called the least fixed point.

Definition 2.13 (Least fixed point)**.** Let $(\sqsubseteq, D)$ be a po. An element $d \in D$ is a least fixed point (lfp) of the function $f\colon D \to D$, if it is a fixed point of f and $d \sqsubseteq x$ for all other fixed points $x \in D$.

The lfp is, thus, the unique least element in the set of fixed points. In general, it is unclear if the lfp exists for a given function on a po and how to obtain it. However, monotonic functions always have such a unique least fixpoint [Tar55]. In the special case of a continuous function mapping a pcpo to itself, the lfp can even be constructed by the so-called bottom iteration as the following theorem shows.

Theorem 1 (Kleene Fixed-Point Theorem [Kle52])**.** Let $(D, \sqsubseteq)$ be a pcpo and $f\colon D \to D$ a continuous function. Then f has an lfp given by

$$\operatorname{lfp}(f) = \bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}.$$

Proof. This proof can be found in [SHLG94, p. 24]. It is repeated here for its elegancy. First, we show that $f^n(\bot) \sqsubseteq f^{n+1}(\bot)$ for all $n \in \mathbb{N}$ using induction. For the base case, let $n = 0$. Then $\bot = f^0(\bot) \sqsubseteq f(\bot)$ since $\bot$ is the least element of D. Henceforth, we may assume $f^{n-1}(\bot) \sqsubseteq f^n(\bot)$ for $n > 0$. As f is continuous, it is also monotonic. Thus, we find $f(f^{n-1}(\bot)) \sqsubseteq f(f^n(\bot))$. Rearranging yields $f^n(\bot) \sqsubseteq f^{n+1}(\bot)$ which concludes the induction. It follows that $\{f^n(\bot) \mid n \in \mathbb{N}\}$ is an ascending chain with least element $\bot$ in D. As D is a pcpo the lub of the chain must exist. Next, we prove that $\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$ is a fixed point of f. Note that removing $f^0(\bot)$ from the chain

does not influence its lub since $\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\} = \bigsqcup\{f^{n+1}(\bot) \mid n \in \mathbb{N}\}$. Given the above observation applying the continuous function f to the lub yields

$$\begin{aligned} f\left(\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}\right) &= \bigsqcup f(\{f^n(\bot) \mid n \in \mathbb{N}\}) \\ &= \bigsqcup\{f^{n+1}(\bot) \mid n \in \mathbb{N}\} \\ &= \bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}. \end{aligned}$$

Hence, the lub is a fixed point of f. At last, we show that $\bigsqcup\{f^n(\bot) \mid n \in \mathbb{N}\}$ is the least fixed point of f. By definition of the lub, it suffices to prove that the set of fixed points of f is a subset of the set of upper bounds of $\{f^n(\bot) \mid n \in \mathbb{N}\}$. As the lub is a fixed point of f as shown above, it must be the least element in the set of upper bounds as well. More specifically, given a fixed point of f, $x \in D$ say, we prove, again via induction, that $f^n(\bot) \sqsubseteq x$ for all $n \in \mathbb{N}$. First, let $n = 0$. Then, $\bot = f^0(\bot) \sqsubseteq x$ as $\bot$ is the least element of D. Hence, we may assume $f^{n-1}(\bot) \sqsubseteq x$ for $n > 0$. Again, f is monotonic as it is continuous. Thus, applying f to the induction hypothesis yields $f(f^{n-1}(\bot)) \sqsubseteq f(x)$. Since x is a fixed point of f the claim follows, which concludes the proof. □

2.3 Isabelle/HOL

Isabelle is an interactive theorem prover developed at the Technical University Munich and the University of Cambridge. It combines user-guided proof construction with automation, allowing for the formal verification of mathematical theorems and software correctness. Isabelle supports multiple logic systems. Isabelle/HOL (for higher-order logic) is the most widely used variant. It includes features like the Isar proof language for readable proofs and Sledgehammer, which integrates external automated provers for automated proving. Researchers, educational institutions, and industrial institutions use Isabelle for tasks such as verifying cryptographic protocols, critical systems, the correctness of an OS-Kernel [KEH+09], and mathematical proofs. The Archive of Formal Proofs[2] is a collection of Isabelle proofs. There are a variety of libraries available that extend Isabelle's pretty small logical core. One of the most frequently used libraries is Isabelle/HOL [NPW02] introduces the concept of higher-order logic as well as data structures like sets, and lists.

Isabelle's Type System

In Isabelle, all variables are typed. Polymorphic types can be denoted via formal type parameters like `'a`. Although Isabelle can in most cases automatically determine the type of a variable or constant `x`, we can also fix the type to a specific one which is

[2] `https://www.isa-afp.org`. Accessed 03.11.2024

denoted as `(x::<typename>)`. The type of a total function with n input parameters of types $\tau_1, \ldots, \tau_n$ and return type τ is denoted as $\tau_1 \Rightarrow \tau_2 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \tau$. As usual, $\Rightarrow$ associates to the right. For instance, the type of the identity function is `'a ⇒'a`. The Isabelle/HOL theories provide some predefined types used for implementing the FOCUS framework in Isabelle, e.g., `nat` for the natural numbers and `bool` for Boolean values. Furthermore, predefined type constructors, i.e., `list`, `set`, and `prod` (tuple type), can be used in postfix syntax to create composite types, such as `nat list` or `bool set`. In addition to the types, helpful functions are provided. The mapping functions `map` for lists, `image` for sets (e.g, `f ` S`), and `apfst` and `apsnd` for tuples are often used in the later sections.

Defining Types

We can also create our own data types in Isabelle. A simple data type can be defined using the syntax shown below:

```
datatype Operator = Plus | Minus
```

The statement above declares the new data type `Operator` with exactly two constructors separated by a | character: `Plus` and `Minus`. Both constructors are nullary, i.e., they do not have any parameters. Thus, their type is `() ⇒ Operator`. Note that the `Operator` type viewed as a set consists of exactly two elements. However, a **`datatype`** based type definition does not instantiate an order on the data type elements. Data types can also be parameterized using formal type parameter `'a` as follows:

```
datatype 'a Box = EmptyBox | Wrap 'a
```

To use this `Box` data type, the formal type parameter `'a` must be instantiated first. For instance, `Wrap (Suc 0)` is of type `nat Box`. Here, `EmptyBox` is a nullary constructor, and `Wrap` is a unary constructor. Note that for a type τ with n values, the type τ `Box` has exactly $n + 1$ values, i.e., `bool Box` has three. We can declare recursive data types by using the declared type on the right-hand side of the type definition:

```
datatype 'a strictlist = Empty | Prepend "'a" "'a strictlist"
```

Again, this generates two constructors with the following types:

```
Empty   :: () ⇒ 'a strictlist
Prepend :: 'a ⇒ 'a strictlist ⇒ 'a strictlist
```

To construct instances of custom data types more concisely, we can define constructor abbreviations in the data type declaration:

```
datatype 'a strictlist = Empty ("[]") |
                         Prepend "'a" "'a strictlist" (infixl "@")
```

This way, we can denote an empty list as `[]` and write `x @ xs` instead of `Prepend x xs`. Notice that **`infixl`** declares `@` as a left-associative infix operator, i.e.

```
x₁ @ x₂ @ ... @ xs = ((x₁ @ x₂) @ ... ) @ xs
```

Lastly, we can name formal constructor parameters such that Isabelle generates selectors for them:

```
datatype 'a strictlist =
  Empty ("[]") |
  Prepend (head :: "'a") (tail :: "'a strictlist") (infixl "@")
```

In this case, Isabelle creates the following two selectors in addition to the infix operator `@`:

```
head :: 'a strictlist ⇒ 'a
tail :: 'a strictlist ⇒ 'a strictlist
```

So when `xs` is a non-empty list of type `'a strictlist`, we can access the first list element via `head xs` and the rest of the list with `tail xs`.

Function and Class Definitions

To define simple non-recursive functions in Isabelle, we can use the **definition** command. For instance, a generic identity function can be defined as shown below:

```
definition id :: "'a ⇒ 'a" where
"id ≡ (λx. x)"
```

Alternatively, if we just want to use such a definition to abbreviate a more complex formula, we can use the **abbreviation** keyword. On one hand, an abbreviation has the advantage that it is automatically replaced by its definition and vice versa, if necessary. On the other hand, this automatic replacement might not always be desired since it can negatively influence Isabelle's proof strategies like the simplifier. Recursive or pattern-matching-based functions can be defined using the **fun** or **primrec** command. For instance, a function that delivers 1 if applied to 0 and otherwise behaves like the identity function can be formalized as shown below:

```
fun succ_zero :: "nat ⇒ nat" where
"succ_zero 0 = 1" |
"succ_zero x = x"
```

As we can see, the syntax of such definitions is similar to Haskell. The same also holds for class definitions. The important difference is, that classes in Isabelle allow us to specify properties of functions. A class for types with an equality function can for example be specified as follows:

```
class Eq =
  fixes myEq :: "'a ⇒ 'a ⇒ bool"
  assumes reflexivity:  "myEq a a = True"
  assumes symmetry:     "myEq a b = myEq b a"
  assumes transitivity: "myEq a b ∧ myEq b c ⟶ myEq a c"
```

An overview of different classes used is given in Table 2.1

In the following definition `isFixpoint` the type in the signature of the definition is restricted to types that have an equality operator.

```
definition isFixpoint ::"('a::{Eq} => 'a) => 'a => bool" where
"isFixpoint f a ≡ f a = a"
```

Name	Assumptions
enum	type can be defined as a distinct, finite list
finite	type has only finitely many elements
countable	type is countable
po	partial order (see Section 2.2)
cpo	complete partial order (see Section 2.2)
pcpo	pointed complete partial order (see Section 2.2)

Table 2.1: Classes overview

Domains in Isabelle

In the previous sections, we saw how the **datatype** constructor can be used to define custom data types. However, such data type definitions have some limitations, i.e., they only consist of values that can be constructed with finitely many applications of the constructors. Furthermore, the **datatype** command does not establish an order on the data type. However, orders are necessary for inductive reasoning over the data type. In the following, we will present three ways to overcome these limitations, namely the lifting of data types, the domain constructor, and subtypes.

Lifting Datatypes to Domains

For any ordinary HOL type, we can define a trivial complete partial order by giving it a discrete ordering. In Higher Order Logic of Computable Functions (HOLCF) this construction is formalized using the `'a discr` type [Huf12]:

```
datatype 'a discr = Discr "'a"
```

To still be able to access the elements of the lifted data type, an inverse of the `Discr` constructor is defined as shown below:

```
definition undiscr :: "'a discr ⇒ 'a" where
"undiscr x ≡ (case x of Discr y ⇒ y)"
```

The ordering on `'a discr` is defined as a flat ordering, i.e., $(x \sqsubseteq y) = (x = y)$. Thus, `'a discr` is an instance of the `discrete_cpo` class [Huf12]. It follows straightforwardly by the corresponding definitions that every function `f :: 'a discr ⇒ 'b` is continuous and every predicate `P :: 'a discr ⇒ bool` is admissible. Furthermore, we can also lift a given type with a complete partial ordering to a type with a pointed cpo by adding a new bottom element. Let D be a cpo (which may or may not have a least element), then the lifted pcpo $D_\perp$ consists of a bottom element $\perp$ and wrapped elements of the original type. In HOLCF, the lifting of cpos to pcpos can be achieved by using the `'a u` type which is also often abbreviated as $'a_\perp$:

```
datatype 'a u = lbottom | lup 'a
```

The order of this data type is defined such that the following holds:

$$a \sqsubseteq b \Leftrightarrow (a = \texttt{lbottom}) \vee (\exists x, y.\ a = \texttt{lup x} \wedge b = \texttt{lup y} \wedge x \sqsubseteq y)$$

One can show that the type $'a_\perp$ is a pcpo, if the type `'a` is substituted with, has a partial order.

The Domain Type-Constructor

In Section 2.2 we already explained how domains can be constructed from simpler domains using domain constructors. To achieve the same in Isabelle, we can use the domain package [Huf12] which produces data types that are instances of the pcpo class and hence have a pcpo ordering. The syntax of such a data type definition is similar to a type definition using the **`datatype`** keyword. However, the domain package defines the data type constructors as strict continuous functions and automatically adds a bottom element $\perp$. An example of such a domain definition is the lazy natural number data type:

```
domain 'a lnat = lnsuc (lazy lnpred::"'a list")
```

As mentioned earlier this also automatically adds a bottom element $\perp$ to the data type and establishes a pointed complete partial order $\sqsubseteq$. Furthermore, the `lnpred` destructor is defined which serves as an inverse function of the `lnsuc` constructor function. The **`lazy`** keyword makes the constructors non-strict in specific arguments.

To facilitate proofs that involve such **`domain`** types, Isabelle also adds several rewrite rules to Isabelle's simplifier (`simp`), and generates the necessary theorems and functions for case distinctions and induction proofs. [Huf12] provides a good overview of all theorems and functions that are automatically generated by the domain package.

CPOs on Subtypes

An even simpler way to create a cpo type is to define it as a subset of an existing cpo type. Under certain conditions, a subtype can inherit the ordering structure from the

existing ordering it is based on which means that the subtype is again a member of the cpo class. In Isabelle, this process can be automated using the **pcpodef** and **cpodef** commands [Huf12]. Both commands are based on the **typedef** command [NPW02] that allows defining a new type as an isomorphic and nonempty subset of an existing type. For example, we can define a new type `zeroToFive` that is isomorphic to the set of all integers that are smaller or equal to 5:

```
typedef zeroToFive = "{x::int. 0 ≤ x ∧ x ≤ 5}"
```

The proof is necessary since we must prove that the newly created types are non-empty. After showing the set on the right side of the definition is non-empty, the **typedef** pack-

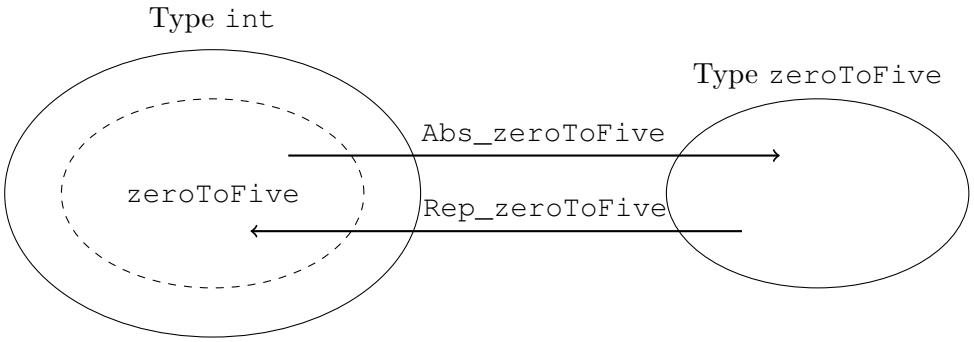

Figure 2.3: Graphical representation of the `typedef` mechanism. [BKR+20]

age automatically creates useful theorems as well as the functions (`Rep_zeroToFive`, `Abs_zeroToFive`) to convert elements of `zeroToFive` to elements of the `int` data type and vice versa. We can then use those functions to define new functions on `zeroToFive` based on existing functions on the `int` type:

```
definition zeroToFive_add:: "zeroToFive ⇒ zeroToFive
                                        ⇒ zeroToFive" where
"zeroToFive_add x y = Abs_zeroToFive (Rep_zeroToFive x
                                    + Rep_zeroToFive y)"
```

The newly defined addition operator on the new type relies on (and hides) the primitive + operator on integers. When using `list_definition` instead of definitions, the conversion of elements for subtypes can be abstracted away. Instead, an additional proof obligation must be fulfilled to show that the result is part of the subtype (e.g., is in-between 0 and 5 for `zeroToFive`).

```
lift_definition zeroToFive_cript:: "zeroToFive ⇒ zeroToFive
                                               ⇒ zeroToFive" is
"λ x y. x * y mod 6"
```

Later, this principle helps to reach an important achievement of this work; creating a high-level API to hide the low-level domain-theoretical concepts from the user. The

lemmata generated by the **typedef** package can also be used to show properties like the commutativity of this function. The **cpodef** and **pcpodef** commands, which automatically construct an ordering on the new subtype, have an identical syntax as **typedef**. If we want to use **cpodef** to define the `zeroToFive` type, we additionally would have to show that predicate λ `x. x` $\leq$ `5` is admissible. In case we want to use **pcpodef** we additionally would have to show that the subtype has a least element.

Continuous Functions and Fixed Points

As we already saw in Section 2.2, the concepts of monotonicity and continuity play an important role in the field of function domains.

Continuous functions in HOLCF are formalized by using the `cfun` data type which is instantiated using the **cpodef** command:

```
cpodef ('a, 'b) cfun ("(_ →/ _)") =
  "{f::'a => 'b. cont f}"
```

Thus, the type of continuous functions from A to B is denoted as `A→B`. To automatically lift anonymous functions to their continuous counterparts, the small letter λ in the definition of such a function can be replaced by Λ. However, such a lifting is of course only successful if the function that should be lifted is continuous. The representation of a continuous function is obtained by applying to the lifted function. Based on the `cfun` type, functions like the **fix** operator which calculates the lfp [MNvOS99] can be defined:

```
primrec iterate :: "nat ⇒ ('a::cpo → 'a::pcpo) → ('a → 'a)" where
"iterate 0 = (Λ F x. x)" |
"iterate (Suc n) = (Λ F x. F·(iterate n·F·x))"

definition fix :: "('a ::pcpo → 'a) → 'a" where
"fix = (Λ F. ⨆i. iterate i·F·⊥)"
```

As we can see, this operator is directly based on the fixed point theorem by Kleene It should also be noted, that the restriction of the type variable `'a` to members of the pcpo class is essential as otherwise the existence of the bottom element that is required in the definition cannot be guaranteed.

Proofs in Isabelle

Isabelle allows formalizing and proving mathematical statements (lemmata) about types, functions, and their behavior. Those proofs are then automatically checked by Isabelle. The following function `succ_zero` maps an input natural number to a natural number that is always greater or equal. Specifically, it maps 0 to 1 and any other natural number to itself.

```
fun succ_zero :: "nat ⇒ nat" where
"succ_zero 0 = 1" |
"succ_zero x = x"
```

The fact that `succ_zero`'s output is always greater or equal to its input is formalized an proven:

```
lemma succ_zero_le: "x ≤ succ_zero x"
  apply(cases x)
   apply simp
  by simp
```

By successively applying proving rules and methods the claim of the `succ_zero_le` lemma is transformed into a tautology. This proof strategy is also called backward chaining. The x variable in the lemma is free and there are no assumptions, hence, the lemma proofs $\forall x.\ x \leq succ\_zero\ x$. Beside the well-known quantifiers $\exists$ and $\forall$, Isabelle provides the universal quantifier $\bigwedge$ to express the notion of an arbitrary value. After successfully proving a fact as a lemma, the fact is directly usable in other proofs. Assumptions can be formalized using the **assumes** and **shows** keywords:

```
lemma succ_zero_eq: assumes "1 ≤ x"
  shows "x = succ_zero x"
  apply(cases x)
  using assms apply auto[1]
  by simp
```

The **shows** keyword separates the actual claim from the assumptions. In proofs, assumptions can then be referenced using `assms` keyword.

Apply-script proofs as shown above are in general not easily readable, since intermediate steps are not explicitly visible. Backward reasoning may complicate understanding proofs further. To overcome these issues, the proof language Isar [Wen02] was introduced, which allows formalizing more human-readable proofs. For instance, the `succ_zero_le` can be proven with Isar as shown below:

```
lemma succ_zero_ge: "x ≤ succ_zero x"
proof(cases "x = 0")
  case True
  thus ?thesis
    by simp
next
  case False
  thus ?thesis
    by (metis False eq_iff succ_zero.elims)
qed
```

Isar provides the means to efficiently structure and implement large proofs.

2.3.1 Alternative Theorem Provers

Today, there exist a variety of theorem provers. Generally, they are divided into interactive and non-interactive theorem provers. While interactive theorem provers often perform better with regard to automated verification, they have some limitations concerning specification options or verification emphasis' (e.g., verification of termination). Theorem provers also differentiate each other with regard to type systems and theoretical foundations/logic.

Karlsruhe Interactive Verifier (KIV)

The KIV [Rei95] is a theorem prover with algebraic specification capabilities. KIV primarily focuses on the refinement-based development of both sequential and concurrent software systems. Recent advancements include support for polymorphism and exceptions in programs. It also showcases the proof engineering support utilizing a graphical user interface and explicit proof trees, along with KIV's assistance in developing large-scale software systems through modular components and verifying concurrent algorithms using a assumption-guarantee calculus. KIV offsprings are further developed, e.g., [SBBR22] based on first-order dynamic logic and the successor KeY [ABB+16] is actively developed with the ambition to integrate the formal software development for Java into industry.

F*

F* is a functional programming language with dependent types and a Theorem Prover developed by Microsoft Research and Inria [SCF+11]. It is in general an automated theorem prover, but extensions allow interactive theorem proving [MAD+19]. F* code can be exported to OCaml, F#, C, WASM, and ASM code [SMR23]. Like Isabelle, F* code is organized in modules, which can be imported. F* is a dependently typed language that has a pure type system, i.e., it treats programs and their types within a single syntactic class. Project Everest is a project to develop a formally verified HTTPS stack driven by Microsoft, Irinia Research, and Carnegie Mellon University. It is composed of various sub-projects including a High-Assurance Cryptographic Library [ZBPB17], a verified cryptographic communication protocol Signal* [PBMB19], and a parser that generates C code from formally proven F* code [RDLF+19].

Dafny

Dafny is a programming language and a static program verifier [Lei10], originally developed by Leino at Microsoft Research in 2008, with its first stable release in 2022. Unlike Isabelle, which uses a different language for proofs than for executable code (physical code), the Dafny language is used for both. Dafny code is only used for proofs (ghost

code) and ghost variables are simply ignored at the code generation phase by the compiler. Fining counterexamples is also possible, using the SMT solver Z3. Intermediary verification code is generated in the language Boogie 2 [BCD+06]. The Ironclad Apps project utilizes Trusted Computing and Dafny for verification to provide a framework in which the user can verify that servers adhere to a high-level specification. A main drawback for Dafny is the requirement of termination. This is not suitable when proving properties for potentially non-terminating functions, e.g., fixpoints.

Coq

Coq is a well-established interactive theorem prover based on the calculus of inductive construction [BC10] which allows for a dependent type system. It is possible to correct by construction code from proofs, and the calculus of inductive construction can be extended to classical logic by the law of excluded middle. Nevertheless, Isabelle offers more readable proof using the Isar language, can export code, and provides a powerful and general proof finder. The most prominent successful application is the formal verification of a C compiler [Ler09].

Lean

Another rather young theorem prover, which also serves as a functional programming language, is Lean [dMKA+15]. It is based on a calculus of inductive construction variant and offers interactive proving as well as automated proving tools. The language emphasizes the compactness and the definition of types and functions. Lean allows for dependent types that are not allowed in Isabelle. In dependent type theory, types can depend on parameters. It is also possible to export Lean proofs to theorem provers with the same theoretical foundation, e.g., Coq. So far, Lean is mainly used for implementing and verifying mathematical theory, e.g., the theorem of liquid modules [Sch22] or topology theorems for perfectoid spaces [BCM20].

Proof Verification System (PVS)

PVS was developed at SRI International Computer and the first version was available in 1993 [ORS92]. PVS implements typed higher-order logic. It comes with many of the standard types like reals, integers, rationals, booleans, or lists. To build more advanced types like records, recursive data types, or function types, you can use type constructors. PVS also gets extended with predicate subtypes and dependent types. A predicate subtype is a new type that gets constructed based on an existing type. With the help of a predicate, you can filter the elements that satisfy the predicate from all elements of the basis type. PVS' syntax is not extendable. You get a predefined, not changeable set of symbols that are usable as infix operators. An automatic prover tool similar to `sledgehammer` in Isabelle is so far not available for PVS. The NASA Langley Formal

Methods Research Program uses PVS in several project areas for air traffic management [DM07], and formal methods for certification research [CM14].

The implementation of the mathematical framework Focus in Isabelle is presented in the next chapters. We start with the most fundamental type in Focus, i.e., the type of streams.

Chapter 3

Timed Streams and Their Encoding

A stream is an observation of transmitted messages over a communication channel. This observation is either a finite or infinite sequence of messages. A timed stream extends the observation by including information on the progression of time. Streams are a fundamental building block of FOCUS components. Chapter 5 utilizes streams to define the semantics of components.

In Section 3.1 we give a mathematical definition of streams. Then the definition of stream in the theorem prover Isabelle is presented. Following, different kinds of timed streams are introduced. We present in Section 3.2 functions over timed streams. A developer can reuse these functions to quickly define their own properties or lemmata. The chapter ends in Section 3.3 with a discussion of alternatives and related work.

3.1 Datatype Definitions

Every stream possesses a specific *domain* i.e. the set of events or messages possible in that observation that the stream denotes. For instance, the domain would be $\mathbb{N}$ if natural numbers are being transmitted. It is common to use the letter M to represent an arbitrary yet fixed domain. In the case of components that accept input messages and generate output messages, the input and output domains are typically denoted as I and O, respectively. These domains allow us to define the data types of streams. For the mathematical definition we distinguish between finite and infinite streams.

Definition 3.1 (Finite Stream [BKR+20, RR11, BS01b])**.** The set of all finite streams over a domain M is denoted by

$$M^* := \{\langle m_1, m_2, \ldots, m_i\rangle \mid \forall j \in [1, i].\ m_j \in M, i \in \mathbb{N}\}$$

The empty stream $\langle\rangle$ is denoted as ϵ.

Definition 3.2 (Infinite Stream [BKR+20, RR11, BS01b])**.** The set of all infinite streams over M is defined as

$$M^\infty := \{\langle m_1, m_2, \ldots\rangle \mid m_i \in M, i \in \mathbb{N}\}$$

Definition 3.3 (Stream)**.** The set of all streams over M is written as

$$M^\omega := M^* \cup M^\infty$$

Streams can be combined using the infix concatenation operator ($^\frown$).

Definition 3.4 (Concatenation)**.** Given two stream $s1, s2 \in M^\omega$ the concatenation $s1^\frown s2$ is defined as

$$s1^\frown s2 := \begin{cases} \langle m_1, m_2, \ldots, m_i, n_1, n_2, \ldots, n_j \rangle & \text{if } s1 \in M^* \text{ and } s1 = \langle m_1, m_2, \ldots, m_i \rangle \\ & \text{and } s2 \in M^* \text{ and } s2 = \langle n_1, n_2, \ldots, n_j \rangle \\ \langle m_1, m_2, \ldots, m_i, n_1, n_2, \ldots \rangle & \text{if } s1 \in M^* \text{ and } s1 = \langle m_1, m_2, \ldots, m_i \rangle \\ & \text{and } s2 \in M^\infty \text{ and } s2 = \langle n_1, n_2, \ldots \rangle \\ s1 & \text{if } s1 \in M^\infty \end{cases}$$

Example 3.1 (Concatenation)**.** *The following examples show usages of the concatenation operator:*

- $\langle 1, 2, 3 \rangle ^\frown \langle 4, 5 \rangle = \langle 1, 2, 3, 4, 5 \rangle$
- $\forall s \in M^\omega.\ s ^\frown \epsilon = s$
- $\forall s \in M^\omega.\ \epsilon ^\frown s = s$
- $\forall x_\infty \in M^\infty . \forall s \in M^\omega.\ x_\infty ^\frown s = x_\infty$

The concatenation operator is used to define the prefix order on streams.

Definition 3.5 (Prefix Order [BKR+20])**.** The prefix ordering [Rum96] on streams ($\sqsubseteq$) is defined as:

$$\forall x, y \in M^\omega.\ (x \sqsubseteq y \Leftrightarrow (\exists s \in M^\omega.\ \ x ^\frown s = y))$$

Example 3.2 (Prefix Order)**.** *The following order relations hold on streams:*

- $\langle 1, 2, 3 \rangle \sqsubseteq \langle 1, 2, 3, 4, 5 \rangle$
- $\langle 1, 2, 3 \rangle \not\sqsubseteq \langle 1, 2, 99 \rangle$
- $\forall s \in M^\omega.\ s \sqsubseteq s$
- $\forall s \in M^\omega.\ \epsilon \sqsubseteq s$
- $\forall x_\infty \in M^\infty . \forall s \in M^\omega . x_\infty \sqsubseteq s \Leftrightarrow x_\infty = s$

The order $\sqsubseteq$ is a pcpo on streams. The empty stream ϵ is the bottom element.

Isabelle Definition The stream definition in Isabelle uses the HOLCF [Reg94] extension. With the keyword `domain` a new pcpo is created. We first show the complete definition and then explain each element in more detail.

```
domain 'm::countable stream = lscons (lshd::"'m discr u")
                    (lazy srt::"'m stream")  (infixr "&&" 65)
```

The type parameter `'m` corresponds to the previously introduced domain M of the stream. For example, a stream over natural numbers has the type `nat stream`. An additional condition of the Isabelle implementation is the countability of `'m`. This is due to restrictions in HOLCF'11 [Huf12], which internally maps the new datatype to a universal domain. Previous versions of HOLCF [MNvOS99] used axioms to define the domain datatype. However, these axioms can lead to paradoxes, as shown in [Huf12]. The stream datatype consists of two constructors. A bottom-constructor (`⊥::'m stream`) is automatically added to each **`domain`** datatype and hence not explicitly defined. This constructor creates the least element. For streams, we also abbreviate this element with ϵ. The second constructor is explicitly defined. It prepends an element to a given stream. Due to conditions in HOLCF, the element must first be converted into a pcpo. This is assured with the type `'m discr u`. To convert the value `x::'m` the constructors `up` and `Discr` are used, e.g, `up·(Discr msg)::'m discr u`. The full signature of the second constructor is `lscons::'m discr u →'m stream →'m stream`. The `lscons` constructor can be abbreviated as infix `&&`. The domain datatype also introduces the selectors `lshd::'m stream →'m discr u` and `srt:: 'm stream →'m stream`. The `lscons` constructor is strict on the first argument but lazy on the second argument. The strictness of the first argument ensures that the message is not $\bot$ while the laziness of the second argument enables infinite streams.

```
lemma "⊥ && s = ϵ"       (* first argument of lscons is strict *)
  and "msg ≠ ⊥ ⟹ msg && ϵ ≠ ϵ" (* second argument is lazy *)
```

Instead of `up` and `&&`, we often construct streams from lists. This is abbreviated as `<ls>` where `ls` is a list:

```
lemma "<[1,2]> = up·(Discr 1) && up·(Discr 2) && ϵ"
```

We encode time progression information in the stream via the `'m` type parameter. For example, `nat event stream` is the stream over natural numbers *with event timing* [RR11]. This way, different kinds of timed streams are typed differently and the type-checker can aid development, e.g., by ensuring that a timed stream cannot be concatenated with an untimed stream. Furthermore, general functions over `'m stream` can directly be used over `'m event stream`, and don't have to be newly defined.

3.1.1 Event Streams as General Form of Timed Streams

The most general timed stream is the *event stream*. Like an untimed stream an event stream is also an observation of transmitted messages over a communication channel.

Additionally, event streams contain information about time. For event streams this timing information is encoded using the additional message `Tick`. A `Tick` observation means a time slice has ended. The duration of a time slice has to be globally defined outside the stream itself, e.g., a time slice could equal one minute or a nanosecond. The messages in a time slice are ordered, but the exact timing of the message is not part of the event stream. An event observation is encoded in the datatype `event`

```
datatype 'm event = EventMsg 'm | Tick ("√")
```

The timeline in Fig. 3.1 shows an example of the event stream ⟨`EventMsg A, EventMsg B, Tick, EventMsg C, Tick, Tick`⟩. The first time slice contains the messages `A` and `B`, the second time slice only the message `C`, and the third time slice is empty. As an abbreviated notation we omit the constructor `EventMsg` when the datatype is clear from the context. In abbreviated notation the stream from Fig. 3.1 is written as $\langle A, B, \surd, C, \surd \rangle$.

Figure 3.1: Overlaying real-world communication events A, B, and C over three discrete time slices. The event stream representation of this communication history is ⟨`EventMsg A, EventMsg B, Tick, EventMsg C, Tick, Tick`⟩.

Event streams are capable of representing incomplete time slices. Take for example the stream ⟨`EventMsg 1`⟩. It consists of an incomplete time slice, since no `Tick` occurs. Incomplete time slices can be used to get a more fine grained view of the processing of components. However, there is a downside. A stream without time but with infinite events can be represented, e.g., the infinite repetition of ⟨`EventMsg 1`⟩. Such streams are not valid observations of inter-components communication and lead to problems when describing components. We thus prohibit event streams from containing infinitely many messages within a single time slice. This so called *wellformedness* property will be further discussed in Section 3.2.1.

Note that time progression in event streams is only as granular as the chosen time interval between Ticks. While the order of messages within a time slice is known, the exact time-stamps are not. The timeline in Fig. 3.2 shows three time slices, all with the events A and B. Each time slice's representation as event stream is ⟨`EventMsg A, EventMsg B, Tick`⟩. While the event stream represetations are equal however, the time slices in the physical world are different due to the different real-world timing of the events.

If a more fine granular view with more information is required, the time slice-duration

Figure 3.2: Overlaying repeated communication events A and B with different real-world timing over discrete time slices. The event stream representation is ⟨`EventMsg A, EventMsg B, Tick, EventMsg A, EventMsg B, Tick, EventMsg A, EventMsg B, Tick`⟩

can be modified. Figure 3.3 displays the same messages as before, but on a 10-times finer timescale. Now there is at most one message in each time slice.

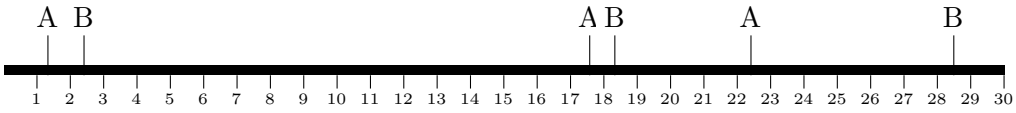

Figure 3.3: Overlaying the events of Fig. 3.2 over a 10-times finer timescale. The event stream representation is ⟨`Tick, EventMsg A, Tick, EventMsg B, Tick, Tick, Tick, Tick, Tick, Tick, Tick, Tick, Tick, Tick, Tick, Tick, Tick, Tick, EventMsg A, Tick, EventMsg B, Tick, Tick, Tick, Tick, EventMsg A, Tick, Tick, Tick, Tick, Tick, Tick, EventMsg B, Tick, Tick`⟩

3.1.2 Time Synchronous Streams

Time synchronous (TSyn) streams are more restrictive than event streams. There is at most one message in each time slice of a time synchronous stream. Hence, a time slice either contains a message or is empty.

```
datatype 'm tsyn = TSynMsg 'm | Eps ("⌣")
```

Figure 3.4 shows an example of a time synchronous Stream. A common application area for time synchronous streams are hardware components, e.g., a computer processor. There a message is not a discrete digital event, instead it is a signal with a duration of one or multiple time slices. It is important to note that the time slice duration and signal duration are synchronized, and a signal over several time slices is to be encoded as multiple messages. Figure 3.5 shows a time synchronous stream with signals. In

Figure 3.4: Example of time synchronus stream ⟨*TSynMsg A, TSynMsg B, Eps*⟩

the entire first time slice the signal A is transmitted. For the mathematical interpretation this difference between discrete message and analog signal is not important. Both observations are represented in the same stream ⟨`TSynMsg A, TSynMsg B, Eps`⟩. The main difference is that synchronous hardware has usually no delay between input and output except when used with feedback, while event-based specifications are often delayed, because one doesn't know how late a message arrives in a time slice.

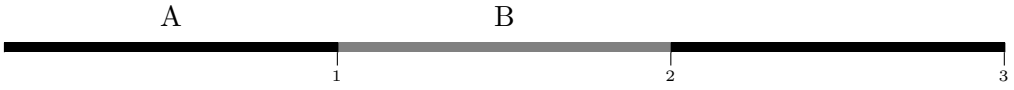

Figure 3.5: Stream from Fig. 3.4 with periods instead of discrete events

Similar to event streams it is possible to increase the granularity of observation. In this case, empty observations `Eps` are added, see Fig. 3.6.

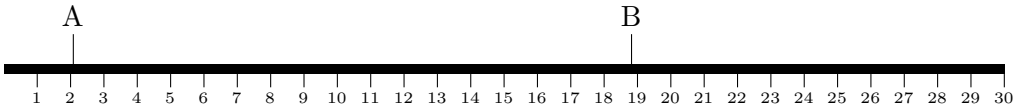

Figure 3.6: Stream from Fig. 3.4 on a 10-times finer time-view

For streams refining the time granularity to smaller time slices is possible, while making it more coarse in general may introduce invalidity. When two consecutive time slices both contain a message, it is impossible to combine them into one time slice since only one message per time slice is allowed.

3.1.3 TOne Streams

If a time synchronous stream always contains a message, the datatype to model such a stream becomes even simpler. TOne streams model a stream where the basic assumption is that we observe exactly one message per unit of time.

```
datatype 'm tone = TOneMsg 'm
```

For technical reasons and to simplify the modeling, the knowledge that it is a TOne stream was encoded in the messages, as with previous stream forms, but not in the stream, by defining an essentially empty wrapper `tone`. This encoding is necessary because otherwise untimed streams like `nat stream` and TOne streams such as `nat tone stream` would not be distinguishable. Technically, it is easier to pack this encoding into the messages (here using the otherwise empty encoding `tone`) than into the streams, because this way we only need a single stream theory. However, we are aware that this is somewhat unintuitive.

Figure 3.7: TOne Stream ⟨*TOneMsg A*, *TOneMsg B*, *TOneMsg C*⟩

Similar to the previous time synchronous streams, TOne streams are often used to represent signals with duration instead of digital messages. Figure 3.8 shows an example of such a stream.

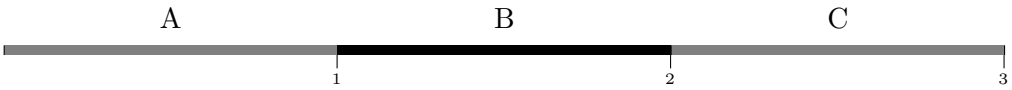

Figure 3.8: Stream from Fig. 3.7 with periods instead of discrete events

Neither increasing nor decreasing time granularity is generally possible for TOne Streams. Similar to time synchronous streams, decreasing the granularity could lead to multiple messages in one time slice, which is not allowed in TOne streams. Increasing the time granularity would lead to empty time slices, which is also not supported by TOne stream.

3.1.4 Comparison of Timed Stream Types

After introducing three different kinds of timed streams, we compare benefits and downsides of each kind. Table 3.2 given an overview of the relevant properties.

Abbrev	Isabelle Type	Description	See Page
$\mathbb{B}$	`bool`	$\{True, False\}$	
$\mathbb{N}$	`nat`	$\{0, 1, 2, 3, \dots\}$	
$\mathbb{N}_\infty$	`lnat`	$\mathbb{N} \cup \{\infty\}$	
$A \times B$	`(A,B) prod`	tuple of elements	
$\mathbb{P}(I)$	`I set`	set	
$M_\perp$	`M discr u`	lift to pcpo	
M^{ev}	`M event`	event datatype: $M \cup \{\surd\}$	30
$M^?$	`M tsyn`	tsyn datatype: $M \cup \{\sim\}$	31
M^{tone}	`M tone`	tone datatype: M	33
M^*	`M list`	finite list	
M^ω	`M::{countable} stream`	general stream	29
$M^{\underline{\omega}}$	`M::{countable,tlen} stream`	general timed stream	
$M^{\underline{*\omega}}$	`M::{countable} event stream`	event stream	30
$M^{\underline{?\omega}}$	`M::{countable} tsyn stream`	time-synchronus stream	31
$M^{\underline{1\omega}}$	`M::{countable} tone stream`	tone stream	33
$M^{\underline{[\omega]}}$	`M::{countable} list stream`	tsliced stream	51
$M^{\underline{\cup\omega}}$	`M::{countable} combinedstream`	combined stream	52
I^Ω	`I::{chan} sb`	general bundle	60
$I^{\underline{\Omega}}$	`I::{tchan} sb`	general timed bundle	60
$I \Rightarrow O$	`(I,O) fun`	function	
$I \to O$	`(I::cpo,O::cpo) cfun`	continuous function	
$I \rightarrow\!\!\blacktriangleright O$	`(I::chan sb,O::chan sb) cfun`	deterministic component	
$I \rightarrow\!\!\blacktriangleright^{\mathtt{t}} O$	`(I::tchan sb,O::tchan sb) cfun`	deterministic, timed component	
$I \rightarrow\!\!\blacktriangleright^{\mathtt{t}}_{\mathtt{w}} O$	`(I::tchan sb,O::tchan sb) tspfw`	weakly causal component	77
$I \rightarrow\!\!\blacktriangleright^{\mathtt{t}}_{\mathtt{s}} O$	`(I::tchan sb,O::tchan sb) tspfs`	strongly causal component	77
$I \rhd O$	`(I::chan sb ,O::chan sb) cfun set`	underspecified component	
$I \rhd^t O$	`(I::tchan sb ,O::tchan sb) cfun set`	underspecified, timed component	
$I \rhd^t_w O$	`(I::tchan sb ,O::tchan sb) tspfw set`	underspecified, weakly causal component	87
$I \rhd^t_s O$	`(I::tchan sb ,O::tchan sb) tspfs set`	underspecified, strongly causal component	87
$I \cup J$	`(I::chan,J::chan) union`	set-union of channels	
$I - J$	`(I::chan,J::chan) minus`	set-minus of channels	

Table 3.1: Type-abbreviations

	#Messages in time slice	Incomplete time slice	Per construction wellformed	Finer time	Rougher time
Event	$0 - \infty$	✓	✗	✓	✓
TSyn	$0 - 1$	✗	✓	✓	✗
TOne	1	✗	✓	✗	✗

Table 3.2: Comparison of different stream kinds

$0-\infty$ in Table 3.2 means that the number of messages in a time slice is unbounded, but it may not be infinite. During the design and implementation phases for time-sensitive systems, event streams prove highly advantageous. They ensure maximum freedom and defer premature decisions. As event stream are incrementally constructed, the induction principle becomes particularly useful in proving properties over it, enabling a more comprehensive characterization of components. However, the wellformedness property has to be kept in mind since not every event stream is wellformed by construction. This will be further discussed in Section 3.2.1. We will present a method to prove wellformedness for all interesting components in Section 5.1.3. In this paper we will focus on `event` timing and present these functions in more detail. TSyn and TOne timing can be used for later stages of development, when the maximal frequency of messages is known. An example are hardware circuits where exactly one signal per time slice occurs.

3.2 Important Stream Datatype Functions

In this section we present important mathematical functions over timed streams and the most relevant basic lemmata about these functions. We start with universal definitions, i.e., for any of timing. Then we show timing-specific definitions, per timing type. The definitions for timed streams build upon the definitions for untimed streams from [BKR+20]. An overview of the untimed definitions is given in Table 3.3.

3.2.1 Important Functions over All Timed Streams

The functions described in this section work over all kinds of timed streams. Table 3.4 lists all functions which work over all kinds of timed streams and are independent of the specific type of timing.

Get Number of Time Slices (i.e. the observation duration) To get the number of time slices in a given stream is helpful, e.g., to define wellformedness (Section 3.2.1) and causality (Section 5.1.1). Because getting the number of time slices is also required for bundles (Section 4.3), we define a new class with a function `tlen`. This function takes

	Signature	Description
ε	**sbot**: M^ω	empty stream
$m\&\&s$	**lscons**: $M_\perp \to M^\omega \to M^\omega$	append first element
$s \sqsubseteq l$	**below**: $M^\omega \Rightarrow M^\omega \Rightarrow \mathbb{B}$	prefix relation
$\uparrow$	**sup'**: $M \Rightarrow M^\omega$	construct stream from an element
$s\vert_n$	**stake**: $\mathbb{N} \Rightarrow M^\omega \to M^\omega$	retrieve the first n elements
$s\hat{}s'$	**sconc**: $M^\omega \Rightarrow M^\omega \to M^\omega$	concatenation of streams
	sValues: $M^\omega \to \mathbb{P}(M)$	the set of all messages in stream
	shd: $M^\omega \Rightarrow M$	first element of stream
	srt: $M^\omega \to M^\omega$	stream without first element
$\#s$	**slen**: $M^\omega \to \mathbb{N}_\infty$	length of stream
	sdrop: $\mathbb{N} \Rightarrow M^\omega \to M^\omega$	remove first n elements
$s.n$	**snth**: $\mathbb{N} \Rightarrow M^\omega \Rightarrow M$	n^{th} element of stream
$s \cdot n$	**sntimes**: $\mathbb{N}_\infty \Rightarrow M^\omega \Rightarrow M^\omega$	stream iterated n times
$s \cdot \infty$	**sinftimes**: $M^\omega \Rightarrow M^\omega$	stream iterated ∞ times
	smap: $(I \Rightarrow O) \Rightarrow I^\omega \to O^\omega$	element-wise function application
	siterate: $(M \Rightarrow M) \Rightarrow M \Rightarrow M^\omega$	infinite iteration of function
$A \ominus s$	**sfilter**: $\mathbb{P}(M) \Rightarrow M^\omega \to M^\omega$	filtering function
	stakewhile: $(M \Rightarrow \mathbb{B}) \Rightarrow M^\omega \to M^\omega$	prefix where predicate holds
	sdropwhile: $(M \Rightarrow \mathbb{B}) \Rightarrow M^\omega \to M^\omega$	drop prefix while predicate holds
	szip: $I^\omega \to O^\omega \to (I \times O)^\omega$	zip two streams into one stream
$\alpha.s$	**srcdups**: $M^\omega \to M^\omega$	remove consecutive duplicates
	slookahd: $I^\omega \to (I \Rightarrow O) \to O$	apply function to head of stream
	sfoot: $M^\omega \Rightarrow M$	last elem. of not empty, finite stream
	sprojfst: $(I \times O)^\omega \to I^\omega$	projects the first stream
	sprojsnd: $(I \times O)^\omega \to O^\omega$	projects the second stream
	stwbl: $(M \Rightarrow \mathbb{B}) \Rightarrow M^\omega \to M^\omega$	stakewhile + first violating element
	srtdw: $(M \Rightarrow \mathbb{B}) \Rightarrow M^\omega \to M^\omega$	dropwhile and then remove head
	sscanl: $(O \Rightarrow I \Rightarrow O) \Rightarrow O \Rightarrow (I^\omega \to O^\omega)$	state-based specifications

Table 3.3: Definitions on all kinds of streams including the untimed streams; all are explained in the first book [BKR+20]. Type-abbreviations are explained in Table 3.1

an element of its class, i.e., `tlen`, and returns its number of time slices as a lazy natural number, `lnat`. Additionally, we require that the function `tlen` is monotonic and that there exists an element a whose length is greater than zero, i.e., $tlen\ a \neq 0$.

```
class tlen = po +
  fixes tlen :: "'a::po ⇒ lnat"
  assumes len_mono: "monofun tlen"
  assumes tlen_notzero: "∃a. tlen a ≠ 0"
```

The `tlen` operator is abbreviated as $\#_{\surd}$.

Instantiating this class is similar for the `TSyn` and `TOne` datatype. In both the `TSyn` and `TOne` type a message is equivalent to exactly one time slice. Hence, the `tlen` is always 1.

```
instantiation tsyn:: (type) tlen
begin
  fun tlen_tsyn:: "'a tsyn ⇒ lnat" where
  "tlen_tsyn _ = 1"

  (* [...] *)
```

For event streams only `Tick` messages represent a time slice. Other message do not progress time. Thus, `Tick` messages have the `tlen` of 1, and `EventMsg` the `tlen` of 0.

```
instantiation event::(type) tlen
begin
  fun tlen_event::"'a::type event ⇒ lnat" where
  "tlen_event  Tick       = 1" |
  "tlen_event (EventMsg _)= 0"

  (* [...] *)
```

Now that the time-datatypes are all instances of the `tlen` class, we can define the `tlen` functions over streams. The number of time slices is defined by recursively adding the `tlen` of all elements in the stream.

```
instantiation stream::("{tlen, countable}") tlen
begin
  fixrec tlen_h::"'a::{tlen, countable} stream → lnat" where
  "tlen_h·ε    = ⊥" |
  "tlen_h·(up·msg&&xs) = lAdd·(tlen (undiscr msg))·(tlen_h·xs)"

  definition tlen_stream::"'a::{tlen, countable} stream ⇒ lnat"
     where
  "tlen_stream = Rep_cfun tlen_h"

  (* [...] *)
end
```

The definition uses `ladd`, which adds two lazy natural numbers (lNats). The function `undiscr` is the inverse of `Discr`. Since `Discr` is bijective, `undiscr` is always defined. With `Rep_cfun` a continuous function is converted into a normal function. Since the instantiation assumes that the message is of class `tlen`, the `tlen` function can only be called on timed streams. Trying to call the `tlen` function with an untimed stream as an argument leads to a type error.

	Signature	Description	See Page
$\#_{\surd}$	**tlen**: $M^{\underline{\omega}} \to \mathbb{N}_\infty$	get number of time slices	37
	tsrt: $M^{\underline{\omega}} \to M^{\underline{\omega}}$	remove the first time slice	38
	tsdrop: $\mathbb{N} \Rightarrow M^{\underline{\omega}} \to M^{\underline{\omega}}$	remove first n time slices	38
	tshd: $M^{\underline{\omega}} \to M^{\underline{\omega}}$	get the first time slices	39
	tstake: $\mathbb{N} \Rightarrow M^{\underline{\omega}} \to M^{\underline{\omega}}$	take first n time slices	39
	tsnth: $\mathbb{N} \Rightarrow M^{\underline{\omega}} \to M^{\underline{\omega}}$	get only the nth time slices	39
	wellformed: $M^{\underline{\omega}} \Rightarrow \mathbb{B}$	infinitely many messages require infinite time	40

Table 3.4: Definitions on all kinds of timed streams. Type-abbreviations are explained in Table 3.1

Using the `tlen` function we can define the functions `take`, `drop` and `snth` over time slices. To define these functions we use general functions over streams.

Drop first N-time slices The `sdrop` function from general streams drops the first n message. Hence, we define a new drop-function for time slices. We employ a helper function called `tsrt`. The helper function drops the first time slice by using `srtdw`. The function `srtdw` takes as first parameter a predicate to indicate what messages shall be dropped. Precisely, it drops all messages that fullfil the predicate and also drops the one message after.

```
definition tsrt::"'a::{tlen,countable} stream → 'a stream" where
"tsrt = srtdw (λmsg. #√msg = 0)"
```

To drop multiple time slices `tsrt` is applied multiple times with `iterate`:

```
definition tsdrop:: "nat ⇒ 'a::{tlen,countable} stream →
    'a::{tlen,countable} stream" where
"tsdrop n ≡ Fix.iterate n·tsrt"
```

Take first N-time slices The counterpart to dropping time slices is taking time slices. To get the first time slice we use `stwbl`. The function `stwbl` takes the first messages

which fulfill the provided predicate and the first message which does not fulfills the predicate. All subsequent messages are dropped.

```
definition tshd:: "'a::{tlen,countable} stream → 'a stream" where
"tshd = stwbl (λe. #√e = 0)"
```

For `tsdrop` we applied `tsrt` n-times. This does not work for the `tstake` operator, since taking the first time slice of the first time slice doesn't change anything. Instead, `tsrt` is used in the recursive call.

```
primrec tstake_nc:: "nat ⇒ 'a::{countable,tlen} stream
                         ⇒ 'a stream" where
"tstake_nc    0    _ = ϵ" |
"tstake_nc (Suc n) s = (tshd·s) ^ tstake_nc n (tsrt·s)"
```

The type of the function `tstake_nc` is not continuous because defining a recursive function and proving continuity at the same time is complicated. Instead, we lift the function `tstake_nc` to the continuous function `tstake` in a separate step.

```
definition tstake:: "nat ⇒ 'a::{countable,tlen} stream
                         → 'a stream" where
"tstake n = Abs_cfun (tstake_nc n)"
```

Take Nth time slice The function `tsnth` is designed to retrieve the N-th time slice from a stream of time slices. This operation combines both dropping and taking functionalities to return the desired time slice.

```
definition tsnth :: "nat ⇒ 'a::{tlen,countable} stream
                          → 'a stream" where
"tsnth n ≡ tshd oo (tsdrop n)"
```

Wellformedness Event streams require a wellformedness predicate to restrict impossible observations. This removes zeno behavior where an infinite number of messages is sent in a finite amount of time. The other variants of timed streams are by definition free of zeno problems. For instance, the time synchronous stream has only finite time slices. We define a class of wellformedness over `len` and `tlen` types:

```
class wellformed = tlen + len +
  fixes wellformed::"'a::{tlen,len} ⇒ bool"
  (* [...] *)
```

We then instantiate the `stream` type to be wellformed and defined its wellformedness as follows. Every stream that has infinitely many message must have infinitely many time slices.

```
instantiation stream ::("{tlen, countable}") wellformed
begin
  definition wellformed_stream::"'a stream ⇒ bool" where
  "wellformed_stream s = (#s = ∞ ⟶ #√s = ∞)"

   (* [...] *)
end
```

3.2.2 Functions over Event Streams

Most functions over event streams are defined by internally calling a function over general streams, with Ticks requiring particular attention. When calling a function over general streams, the already proven lemmata over this function directly follow. Defining a completely new function, e.g., as a fixpoint-equation, is more effort and thus seldomly done. All functions over event streams are listed Table 3.5.

Induction Event Streams have two relevant induction methods. The first one is similar to the untimed induction as it inducts over individual messages. The empty stream serves as the base case and in the induction step either Events or Ticks are prepended.

```
lemma eind:
  assumes "adm P"
      and "P ϵ"
      and "⋀msg s. P s ⟹ P (↑(EventMsg msg) ^ s)"
      and "⋀s. P s ⟹ P (↑(Tick) ^ s)"
  shows "P x"
```

The second induction method inducts over time slices. Here the base case are all finite time slices `ls`, which only contain Events and no Ticks. The time slices are typed as lists and thus need conversion to streams using the angled brackets operator. In the induction step a complete time slice is concatenated. This time slice consists of a finite number of events and a Tick.

```
lemma tslice_ind:
  assumes "adm P"
      and "⋀ls. P (smap EventMsg·(<ls>))"
      and "⋀ls s. P s ⟹ P (smap EventMsg·(<ls>) ^ ↑Tick ^ s)"
  shows "P s"
```

Remove Time Information The function `e2ustream` removes all time information of an event stream and returns an untimed stream. This method is often used to define and prove untimed properties over timed streams. First, all `Ticks` are removed from the event stream. Then, the event datatype is removed by applying `(inv EventMsg)` to

Signature	Description	See Page
EventMsg: $M \Rightarrow M^{ev}$	convert message to event message	30
Tick: M^{ev}	end of time slice	30
e2ustream: $M^{*\underline{\omega}} \to M^{\omega}$	convert to untimed stream	41
eValues: $M^{*\underline{\omega}} \to \mathbb{P}(M)$	set of all messages in the stream	42
emap: $(I \Rightarrow O) \Rightarrow I^{*\underline{\omega}} \to O^{*\underline{\omega}}$	element-wise function application	43
efilter: $(M \Rightarrow \mathbb{B}) \Rightarrow M^{*\underline{\omega}} \Rightarrow M^{*\underline{\omega}}$	events filter with given predicate	43
eshdts: $M^{*\underline{\omega}} \to M^{\omega}$	retrieve the first time slice	42
esnth: $\mathbb{N} \Rightarrow M^{*\underline{\omega}} \to M^{\omega}$	retrieve the nth-time slice	42
elenMsg: $M^{*\underline{\omega}} \to \mathbb{N}_{\infty}$	number of events	42
escanlMsg: $(S \Rightarrow I \Rightarrow S \times O^{\omega}) \Rightarrow S \Rightarrow I^{*\underline{\omega}} \to O^{*\underline{\omega}}$	state-based specification ignoring ticks	43
e2rougherTime: $\mathbb{N} \Rightarrow M^{*\underline{\omega}} \to M^{*\underline{\omega}}$	keep only every nth tick	43
e2finerTime: $\mathbb{N} \Rightarrow M^{*\underline{\omega}} \to \mathbb{P}(M^{*\underline{\omega}})$	to finer time, introducing under-specification	44
etakeMsg: $\mathbb{N} \Rightarrow M^{*\underline{\omega}} \to M^{*\underline{\omega}}$	take first n events, keep all Ticks between	44
edropMsg: $\mathbb{N} \Rightarrow M^{*\underline{\omega}} \to M^{*\underline{\omega}}$	drop first n events, keep all Ticks after	44
etakewhile: $(M \Rightarrow \mathbb{B}) \Rightarrow M^{*\underline{\omega}} \to M^{*\underline{\omega}}$	take while condition holds	45
edropwhile: $(M \Rightarrow \mathbb{B}) \Rightarrow M^{*\underline{\omega}} \to M^{*\underline{\omega}}$	drop while condition holds	45
eprojFst: $(A \times B)^{*\underline{\omega}} \to A^{*\underline{\omega}}$	get first element of tuple stream	45
eprojSnd: $(A \times B)^{*\underline{\omega}} \to B^{*\underline{\omega}}$	get second element of tuple stream	45
ezip: $A^{*\underline{\omega}} \to B^{*\underline{\omega}} \to (A \times B)^{*\underline{\omega}}$	zip streams. Drops messages when number of elements in time slice not equal	46

Table 3.5: Definitions on event streams. Type-abbreviations are explained in Table 3.1

all elements. Since the Ticks are already removed, this inverse function is well defined.

```
definition e2ustream::"'a event stream → 'a stream" where
"e2ustream = smap (inv EventMsg) oo sfilter (UNIV -{Tick})"
```

Number of Events To get the number of events, the event stream is first converted into an untimed streams where all messages are events. Then the number of messages is counted with the normal length-operator (#) on general streams.

```
definition elenMsg::"'msg event stream → lnat" where
"elenMsg = (Λ s. #(e2ustream·s))"
```

First Time Slice While the function `tshd` works over all timed streams, there exists an event-specific version. When viewing a single time slice, additional time information is not required. Hence, the single time slice can be an untimed stream. The signature of `tshd` applied on event streams is `'m event stream →'m event stream`. The result is a timed stream. Thus, we define new function over event streams, which returns the first time slice as an untimed stream.

```
definition eshdts::"'a event stream → 'a stream"where
"eshdts = e2ustream oo tshd"
```

Nth Time Slice Same as the first time slice, the nth time slice can also be untimed. To get the nth-time slice, the first n-time slices are dropped and the first time slice of the rest returned. As usual, counting the time slices starts with zero.

```
definition esnth::"nat ⇒ 'a event stream → 'a stream" where
"esnth n = eshdts oo (tsdrop n)"
```

Get Set of Messages Often, only the set of transmitted messages is relevant. For example to ensure that certain messages are never transmitted. The function `eValues` is a continuous functions which returns all messages that occur in the event stream.

```
definition eValues::"'a event stream → 'a set" where
"eValues = sValues oo e2ustream"
```

Stream Mapping The general `smap` applied to timed streams gives access to the timing information, i.e., allows for Ticks to be mapped. This is turn open up the possibility to add or remove Ticks. Often, this is not required as only the events should be mapped. Only being able to map events removes possible error sources and ensures that both input and output stream have the same number of time slices. First, we define a helper function which applies a function `f` to `EventMsg` without modifying `Tick` messages.

```
fun map_event::"('a ⇒ 'b) ⇒ 'a event ⇒ 'b event" where
"map_event _    Tick       = Tick " |
"map_event f (EventMsg a) = EventMsg (f a)"
```

Then the mapping over event streams can be defined using `smap`.

```
definition emap::"('a ⇒ 'b) ⇒ 'a event stream → 'b event stream"
   where
"emap f = smap (map_event f)"
```

Filter Elements Similar to mapping, a dedicated filter operator for event streams should not alter `Ticks`. This ensures that the number of time slices is not modified.

```
definition efilter::"('a ⇒ bool) ⇒  'a event stream → 'a event
   stream" where
"efilter P = sfilter (insert Tick {EventMsg a | a. P a})"
```

State-Based Message Handling This function implements a state-based specification that processes an incoming event stream while ignoring Ticks. The function `escanlMsg` takes three parameters: a function f that defines the state transition, an initial state, and an event stream of messages.

```
definition escanlMsg::
  "('state ⇒ 'msg ⇒ ('state × 'out stream))
     ⇒ 'state ⇒ 'msg event stream → 'out event stream"
```

The core operation of `escanlMsg` utilizes `sscanlA`, which applies the provided function f to each message in the stream, updating the state accordingly. If a Tick is processed, it produces a corresponding Tick in the output without altering the state. For events, it calls the function f with the current state and the event, producing a new state and an output stream. In essence, `escanlMsg` allows for state-based behavior definition while ensuring that Ticks are propagated through the output stream without affecting the underlying state transitions.

Convert to Rougher Time Format The function `e2rougherTime` has the following signature:

```
definition e2rougherTime::
    "nat ⇒ 'msg event stream → 'msg event stream"
```

This function is designed to keep only every nth Tick from an incoming event stream. It takes a natural number n as its parameter, which specifies the interval at which Ticks should be retained. In the implementation of `e2rougherTime`, each Tick is counted. If the current count reaches zero, that Tick is preserved in the output stream, and the count resets to n. If the count has not reached zero, that Tick is ignored and not included in the output. In summary, `e2rougherTime` filters the event stream by retaining only those Ticks that occur at specified intervals while discarding others, thereby providing rougher time-representation.

Convert to Finer Time Format The function `e2finerTime` is defined as follows:

```
definition e2finerTime::
    "nat ⇒ 'msg event stream ⇒ 'msg event stream set"
```

This function returns a set of event streams that correspond to a finer time resolution based on the specified natural number n. It takes an incoming event stream as its second parameter and produces all possible event streams that, when processed through the `e2rougherTime` function with the same n, yield the original stream. The key aspect of `e2finerTime` is that it introduces underspecification regarding where new Ticks may be inserted within the time-slices of the input event stream. Thus, the function returns a set of all possible finer event streams.

Extract First n Events and Preserve Ticks The function `etakeMsg` is defined as follows:

```
definition etakeMsg::
    "nat ⇒ 'msg event stream → 'msg event stream"
```

This function is designed to take the first n events from an incoming event stream while preserving all Ticks that occur between those events. It takes a natural number n as its parameter, which specifies the maximum number of events to retain. In the implementation of `etakeMsg`, the function processes the event stream by counting the number of events encountered. Once n events have been captured, any subsequent events are ignored. Each Tick that appears in the stream is retained. In summary, `etakeMsg` effectively extracts a specified number of events from an event stream while ensuring that all Ticks are maintained.

Drop First n Events and Preserve Ticks The function `edropMsg` is defined as follows:

```
definition edropMsg::
    "nat ⇒ 'msg event stream → 'msg event stream"
```

This function is designed to drop the first nevents from an incoming event stream while retaining all Ticks that occur after those events. It takes a natural number n as its parameter, which specifies the number of initial events to be discarded. In the implementation of `edropMsg`, the function processes the event stream by counting the number of events encountered. Once n events have been skipped, all subsequent events are included in the resulting stream. Each Tick that appears in the stream is retained. In summary, `edropMsg` effectively removes a specified number of initial events from an event stream while ensuring that all Ticks are maintained.

Take While Condition Holds The function `etakewhile` is defined as follows:

```
definition etakewhile::
    "('msg ⇒ bool) ⇒ 'msg event stream → 'msg event stream"
```

This function is designed to take elements from an incoming event stream as long as a specified condition holds true. It accepts a predicate function f that determines whether each element should be retained. In the implementation of `etakewhile`, the function processes the event stream by evaluating each event against the provided condition. If an event meets the condition it is included in the output stream. However, once the condition evaluates to false for any event, no further output is produced. In summary, `etakewhile` extracts elements from an event stream based until the condition fails.

Drop While Condition Holds The function `edropwhile` is defined as follows:

```
definition edropwhile::
    "('msg ⇒ bool) ⇒ 'msg event stream → 'msg event stream"
```

This function is designed to drop elements from an event stream as long as a specified condition holds true. It accepts a predicate function f that determines whether each element should be discarded. In the implementation of `edropwhile`, the function processes the event stream by evaluating each event against the provided condition. If an event meets the condition, it is dropped from the output stream. However, once an event does not meet the condition, all subsequent elements are retained in the output. In summary, `edropwhile` removes elements from an event stream based on a specified condition until that condition fails.

Project First Event The function `eprojFst` is defined as follows:

```
definition eprojFst:: "('a×'b) event stream → 'a event stream"
```

In the implementation of `eprojFst`, the function processes the event stream by applying a mapping operation that retrieves the first component of each tuple. In summary, `eprojFst` retrieves the first tuple-element of an event stream with tuple-messages.

Project Second Event The function `eprojSnd` is defined as follows:

```
definition eprojSnd:: "('a×'b) event stream → 'b event stream"
```

In the implementation of `eprojSnd`, the function processes the event stream by applying a mapping operation that retrieves the second component of each tuple. In summary, `eprojSnd` retrieves the second tuple-element of an event stream with tuple-messages.

Zip Two Event Streams The function `ezip` is defined as follows:

```
definition ezip:: "'a event stream → 'b event stream
                    → ('a×'b) event stream"
```

This function is designed to combine two incoming event streams into a single stream of pairs. In the implementation of `ezip`, the function processes both input streams concurrently and produces pairs of corresponding elements. It requires that the number of messages in each time slice must match; if they do not, events from the longer time slice are dropped to ensure that only complete pairs are produced.

3.2.3 Functions over TSyn Streams

TSyn streams have similar functions to event streams. Functions over TSyn are listed in Table 3.6.

Signature	Description	See Page
TSynMsg: $M \Rightarrow M^?$	convert message to tsyn message	31
Eps: $M^?$	empty time slice	31
tsyn2ustream: $M^{\underline{?\omega}} \to M^\omega$	convert to untimed stream	47
tsynMap: $(I \Rightarrow O) \Rightarrow I^{\underline{?\omega}} \to O^{\underline{?\omega}}$	element-wise function application	47
tsynFilter: $\mathbb{P}(M) \Rightarrow M^{\underline{?\omega}} \to M^{\underline{?\omega}}$	tsyn filter with given predicate	47
snthTsyn: $\mathbb{N} \Rightarrow M^{\underline{?\omega}} \to M^\omega$	retrieve the n-th time slices	47
tsyn2event: $M^{\underline{?\omega}} \to M^{\underline{*\omega}}$	convert tsyn stream to event stream	47
tsynEps: $M^{\underline{?\omega}} \to \mathbb{N}_\infty$	number of empty time slices	48
tsynLen: $M^{\underline{?\omega}} \to \mathbb{N}_\infty$	number of messages	48
tsyntakewhile: $(M \Rightarrow \mathbb{B}) \Rightarrow M^{\underline{?\omega}} \to M^{\underline{?\omega}}$	prefix where predicate holds	48
tsyndropwhile: $(M \Rightarrow \mathbb{B}) \Rightarrow M^{\underline{?\omega}} \to M^{\underline{?\omega}}$	drop prefix while predicate holds	48
tsynProjFst: $(A \times B)^{\underline{?\omega}} \to A^{\underline{?\omega}}$	get first element of tuple stream	48
tsynProjSnd: $(A \times B)^{\underline{?\omega}} \to B^{\underline{?\omega}}$	get second element of tuple stream	48
tsynZip: $A^{\underline{?\omega}} \to B^{\underline{?\omega}} \to (A \times B)^{\underline{?\omega}}$	zip two streams into one stream	49

Table 3.6: Definitions on TSyn Streams. Type-abbreviations are explained in Table 3.1

Remove Time Information `tsyn2ustream` converts a timed synchronic stream into an untimed stream by removing all empty time slices (`Eps`) and retaining only the actual message elements. This transformation is essential for analyzing the content of the stream without considering timing information. The function first filters out any occurrences of `Eps` from the stream, then applies the helper function `tsyn2ustreamElem` to

extract the messages from the remaining time slices. The resulting output is an untimed stream consisting solely of relevant messages.

```
definition tsyn2ustream:: "'a tsyn stream ⇒ 'a stream"
```

Stream Mapping The function `tsynMap` has the following signature:

```
definition tsynMap:: "('a ⇒ 'b) ⇒ 'a tsyn stream ⇒ 'b tsyn stream"
```

This function is designed to apply a given mapping function f to each element of an incoming timed stream while preserving the timing information. Empty time slices are not modified.

Filter Elements The function `tsynFilter` has the following signature:

```
definition tsynFilter:: "'a set ⇒ 'a tsyn stream ⇒ 'a tsyn stream"
```

This function is designed to filter elements from an incoming timed stream based on a specified predicate, represented as a set. In the implementation of `tsynFilter`, the function processes each element in the timed stream and removes those that are not included in the given set. Empty time slices are maintained.

Nth Time Slice The function `snthTsyn` retrieves the N-th time slice from a tsyn stream.

```
definition snthTsyn:: "nat ⇒ 'a tsyn stream ⇒ 'a stream"
```

Convert to Event Stream Any `tsyn` streams can be converted to `event` streams. An empty time slice (`Eps`) is directly mapped to the `Tick`. But `TSynMsg` elements must be mapped to two elements in the event stream. The first elements contains the message `EventMsg m` and the second element the time information `Tick`. We first introduce a helper function that converts individual elements:

```
fun tsyn2eventElem:: "'a tsyn ⇒ 'a event stream" where
"tsyn2eventElem     Eps      = ↑Tick"
"tsyn2eventElem (TSynMsg m) = <[(EventMsg m),Tick]>" |
```

Then we apply this helper using an altered version of `smap`. We cannot use `smap` directly as it can only map to single elements. With `smapAlt` it is possible to map a single element in the input stream to multiple elements in the output stream.

```
definition tsyn2event::"'a tsyn stream ⇒ 'a event stream" where
"tsyn2event ≡ smapAlt tsyn2eventElem"
```

Counting Empty Time slices The function `tsynEps` returns the number of empty time slices:

```
definition tsynEps :: "'a tsyn stream → lnat"
```

Counting Messages The function `tsynLen` returns the number of messages:

```
definition tsynLen:: "'a tsyn stream → lnat"
```

Take While Condition Holds The function `tsyntakewhile` is defined as follows:

```
definition tsyntakewhile::
    "('a ⇒ bool) ⇒ 'a tsyn stream → 'a tsyn stream"
```

This function is designed to take elements from an incoming tsyn stream as long as a specified condition holds true. It accepts a predicate function f that determines whether each element should be retained. In the implementation of `tsyntakewhile`, the function processes the tsyn stream by evaluating each message against the provided condition. If a message meets the condition it is included in the output stream. However, once the condition evaluates to false for any message, no further output is produced.

Drop While Condition Holds The function `tsyndropwhile` is defined as follows:

```
definition tsyndropwhile::
    "('a ⇒ bool) ⇒ 'a tsyn stream → 'a tsyn stream"
```

This function is designed to drop elements from an tsyn stream as long as a specified condition holds true. It accepts a predicate function f that determines whether each element should be discarded. In the implementation of `tsyndropwhile`, the function processes the tsyn stream by evaluating each message against the provided condition. If a message meets the condition, it is dropped from the output stream. However, once an event does not meet the condition, all subsequent messages are retained in the output.

Project First Tsyn The function `tsynProjFst` is defined as follows:

```
definition tsynProjFst:: "('a × 'b) tsyn stream → 'a tsyn stream"
```

In the implementation of `tsynProjFst`, the function processes the tsyn stream by applying a mapping operation that retrieves the first component of each tuple.

Project Second Tsyn The function `tsynProjSnd` is defined as follows:

```
definition tsynProjSnd::"('a×'b) tsyn stream → 'b tsyn stream"
```

In the implementation of `tsynProjSnd`, the function processes the tsyn stream by applying a mapping operation that retrieves the second component of each tuple.

Zip Two Tsyn Streams The function `tsynZip` is defined as follows:

```
definition tsynZip::
    "'a tsyn stream → 'b tsyn stream → ('a × 'b) tsyn stream"
```

This function is designed to combine two incoming tsyn streams into a single stream of pairs. It requires that the number of messages in each time slice must match; if they do not, `Eps` is produced.

3.2.4 Functions over TOne Streams

Functions over TOne are listed in Table 3.7.

Signature	Description	See Page
ToneMsg: $M \Rightarrow M^{tone}$	convert message to tone message	33
tone2ustream: $M^{\underline{1\omega}} \rightarrow M^{\omega}$	convert to untimed stream	49
toneMap: $(I \Rightarrow O) \Rightarrow I^{\underline{1\omega}} \rightarrow O^{\underline{1\omega}}$	element-wise function application	49
toneFilter: $\mathbb{P}(M) \Rightarrow M^{\underline{1\omega}} \rightarrow M^{\underline{1\omega}}$	filter with given predicate	50
snthTone: $\mathbb{N} \Rightarrow M^{\underline{1\omega}} \rightarrow M^{\underline{1\omega}}$	retrieve the nth time slice	50
tone2event: $M^{\underline{1\omega}} \rightarrow M^{\underline{*\omega}}$	convert tone stream to event stream	50
tone2tsyn: $M^{\underline{1\omega}} \rightarrow M^{\underline{?\omega}}$	convert tone stream to tsyn stream	50
tonetakewhile: $(M \Rightarrow \mathbb{B}) \Rightarrow M^{\underline{1\omega}} \rightarrow M^{\underline{1\omega}}$	prefix where predicate holds	50
tonedropwhile: $(M \Rightarrow \mathbb{B}) \Rightarrow M^{\underline{1\omega}} \rightarrow M^{\underline{1\omega}}$	drop prefix while predicate holds	51
tonezip: $A^{\underline{1\omega}} \rightarrow B^{\underline{1\omega}} \rightarrow (A \times B)^{\underline{1\omega}}$	zip two streams into one stream	51
toneprojFst: $(A \times B)^{\underline{1\omega}} \rightarrow A^{\underline{1\omega}}$	get first element of tuple stream	51
toneprojSnd: $(A \times B)^{\underline{1\omega}} \rightarrow B^{\underline{1\omega}}$	get second element of tuple stream	51
tonescanl: $(S \Rightarrow I \Rightarrow S \times O^{\underline{1\omega}}) \Rightarrow S \Rightarrow I^{\underline{1\omega}} \rightarrow O^{\underline{1\omega}}$	state-based specification	51

Table 3.7: Definitions on TOne streams. Type-abbreviations are explained in Table 3.1

Remove Time Information `tone2ustream` converts a tone stream into an untimed stream by retaining only the actual message elements. This transformation is essential for analyzing the content of the stream without considering timing information.

```
definition tone2ustream:: "'a tone stream → 'a stream"
```

Stream Mapping The function `toneMap` has the following signature:

```
definition toneMap:: "('a ⇒ 'b) ⇒ 'a tone stream → 'b tone stream"
```

This function is designed to apply a given mapping function f to each element of an incoming timed stream while preserving the timing information.

Filter Elements The function `toneFilter` has the following signature:

```
definition toneFilter:: "'a set ⇒ 'a tone stream → 'a tone stream"
```

This function is designed to filter elements from an incoming timed stream based on a specified predicate, represented as a set. In the implementation of `toneFilter`, the function processes each element in the timed stream and removes those that are not included in the given set.

Nth Time Slice The function `snthTOne` retrieves the N-th time slice from a tone stream.

```
definition snthTOne:: "nat ⇒ 'a tone stream → 'a stream"
```

Convert to Event Stream Any `tone` stream can be converted to `event` streams.

```
definition tone2event::"'a tone stream → 'a event stream"
```

Convert to Tsyn Stream Any `tone` stream can be converted to `tsyn` streams.

```
definition tone2tsyn::"'a tone stream → 'a tsyn stream"
```

Take While Condition Holds The function `tonetakewhile` is defined as follows:

```
definition tonetakewhile::
    "('a ⇒ bool) ⇒ 'a tone stream → 'a tone stream"
```

This function is designed to take elements from an incoming tone stream as long as a specified condition holds true. It accepts a predicate function f that determines whether each element should be retained. In the implementation of `tonetakewhile`, the function processes the stream by evaluating each message against the provided condition. If a message meets the condition it is included in the output stream. However, once the condition evaluates to false for any message, no further output is produced.

Drop While Condition Holds The function `tonedropwhile` is defined as follows:

```
definition tonedropwhile::
    "('a ⇒ bool) ⇒ 'a tone stream → 'a tone stream"
```

This function is designed to drop elements from an tone stream as long as a specified condition holds true. It accepts a predicate function f that determines whether each element should be discarded. In the implementation of `tonedropwhile`, the function processes the stream by evaluating each message against the provided condition. If a message meets the condition, it is dropped from the output stream. However, once a message does not meet the condition, all subsequent messages are retained in the output.

Project First TOne The function `toneProjFst` is defined as follows:

```
definition toneprojFst:: "('a × 'b) tone stream → 'a tone stream"
```

In the implementation of `toneProjFst`, the function processes the stream by applying a mapping operation that retrieves the first component of each tuple.

Project Second TOne The function `toneProjSnd` is defined as follows:

```
definition toneprojSnd:: "('a×'b) tone stream → 'b tone stream"
```

In the implementation of `toneProjSnd`, the function processes the stream by applying a mapping operation that retrieves the second component of each tuple.

Zip Two TOne Streams The function `toneZip` is defined as follows:

```
definition tonezip::
    "'a tone stream → 'b tone stream → ('a × 'b) tone stream"
```

This function is designed to combine two incoming tone streams into a single stream of pairs.

State-Based Message Handling This function implements a state-based specification that processes a tone stream. The function `tonescanl` takes three parameters: a function f that defines the state transition, an initial state, and a tone stream of messages.

```
definition tonescanl::
  "('state ⇒ 'msg ⇒ ('state × 'out tone stream))
     ⇒ 'state ⇒ 'msg tone stream → 'out tone stream"
```

3.2.5 Complete Time Slices

Another alternative to `event stream` is a timed stream where each time slice is complete. Multiple messages in a time slice are also supported.

The datatype `list stream` has these properties. Since lists are always finite in Isabelle, each time slice is also finite. No additional wellformedness property is required. An example is shown in Fig. 3.9.

Figure 3.9: Example of List Stream ⟨[A,B],[C],[]⟩

It is possible to convert `list stream` to `event stream` and vice versa. Event streams are more fine grained than list stream since they can encode incomplete time slices. Thus the mapping from list stream to event stream is only surjective and not injective. Since incomplete time slices are interesting when modeling components we focus on `event stream` in this work.

3.2.6 Combined Stream

An additional stream type combining the possible timed stream representations is introduced to allow a parametric channel type for stream bundles in Chapter 4 while still enabling the channel-specific restriction of messages and different time modes.

```
datatype 'm combinedstream = TEVENT  "'m event stream" |
                             TSYN    "'m tsyn stream"  |
                             TONE    "'m tone stream"  |
                             TSLICED "'m list stream"  |
                             UNTIMED "'m stream"
```

The previous prefix order over streams is lifted to the `combinedstream` type. Since there are multiple least elements, e.g., `UNTIMED` ϵ and `TEVENT` ϵ, the combined stream type is only a cpo, and not a pcpo. To reuse existing functions specific to each time mode, the `combStreamCases` function lifting the time mode specific functions to a function over `combinedstream` is defined with the following behavior:

```
lemma "combStreamCases f1 f2 f3 f4 f5·(TEVENT s1)  = f1·s1"
  and "combStreamCases f1 f2 f3 f4 f5·(TSYN s2)    = f2·s2"
  and "combStreamCases f1 f2 f3 f4 f5·(TONE s3)    = f3·s3"
  and "combStreamCases f1 f2 f3 f4 f5·(TSLICED s4) = f4·s4"
  and "combStreamCases f1 f2 f3 f4 f5·(UNTIMED s5) = f5·s5"
```

Depending on the time mode, the correct function is applied to the stream. This facilitates simple definitions of the `combValues` function, which gets all messages in a `combinedstream`.

```
definition combValues:: "'m::countable combinedstream → 'm set"
   where
"combValues = combStreamCases eValues tsynValues toneValues
   tslicedValues sValues"
```

The enum `timeType` represents the possible time modes:

```
datatype timeType = TTLive | TTsyn | TTOne | TTFin | TUntimed
```

The straightforward `TUntimed` defines the untimed option, `TTOne` represents the case where there is exactly one message per time slice, `TTsyn` the case with a maximum of one message per time slice, `TTFin` the case with finitely many messages in each time slice and only complete time slices, and `TTLive` the case with arbitrarily many messages in each time slice and possible incomplete time slices.

The `timeType` of a `combinedstream` is obtained using the `combGetTime` function:

```
fun combGetTime::"'m::countable combinedstream ⇒ timeType" where
"combGetTime (UNTIMED s) = TUntimed" |
"combGetTime (TEVENT s)  = TTLive"   |
"combGetTime (TSLICED s) = TTFin"    |
"combGetTime (TONE s)    = TTOne"    |
"combGetTime (TSYN s)    = TTsyn"
```

Finally, the least combined stream of a timing mode is defined as the empty stream with the associated constructor:

```
fun combLeast::"timeType ⇒ 'm::countable combinedstream" where
"combLeast TUntimed = UNTIMED ϵ" |
"combLeast TTOne    = TONE    ϵ" |
"combLeast TTFin    = TSLICED ϵ" |
"combLeast TTLive   = TEVENT  ϵ" |
"combLeast TTsyn    = TSYN    ϵ"
```

The timed length of a combined stream depends on its time representation. The Isabelle implementation uses the `combStreamCases` function internally to apply the correct `tlen` function to the stream.

```
definition tlen_combinedstream::"'m combinedstream ⇒ lnat" where
"tlen_combinedstream = Rep_cfun (combStreamCases (Abs_cfun tlen)
       (Abs_cfun tlen) (Abs_cfun tlen) (Abs_cfun len) undefined)"
```

The `tlen_combinedstream` function applies the correct `tlen` function for the different time modes and is `undefined` for untimed streams. Similar to the other timed length functions over streams, `tlen_combinedstream` is abbreviated as $\#_{\surd}$.

Wellformedness of combined streams is important, since they can be event streams:

```
fun wellformed_combstream::"'msg combinedstream ⇒ bool" where
"wellformed_combstream (TEVENT s) = wellformed s"    |
"wellformed_combstream _ = True"
```

For all other timed or untimed cases, the combined stream is automatically wellformed.

3.3 Alternative Definitions and Discussion

This section shows alternatives to event streams and discusses benefits and downsides.

Infinite Timed Streams by Manfred Broy

Broy [BS01b] defines timed streams as infinite observations. Thus every timed stream has infinitely many ticks. Mathematically, this kind of stream is defined as a function:

$$M_{Broy}^{\infty} := \mathbb{P}(\mathbb{N} \to M^*)$$

For $f \in M_{Broy}^{\infty}$ the result of $f(n)$ is the list of messages in the n-th time slice. Since all time slices are infinite, there exists neither prefix order nor directly induction. In comparison, our `event stream` additionally also models finite and even incomplete timeslices. To define certain properties over components, Broy later also introduces finite timed streams, but only as auxiliary structure. This is discussed in more detail in Section 5.1.1. Both forms of models have their individual benefits. For our purposes, we found that a single datatype for both finite and infinite streams is preferable. For one, we frequently use induction over our streams to prove properties, which needs finite prefixes for the induction principle. Furthermore, functions only have to be defined once over our streams, and work both for finite and infinite streams.

Dense Stream by Manfred Broy

All of the previously discussed timings are discrete. Furthermore, the exact time of a message is lost in the encoding and only a finite number of messages is allowed in a time slice. These assumptions are adequate when modeling the interactions between computer systems or transmitting concrete items, since these interactions are discrete. To fully model analogue continuous behavior, e.g., for cyber physical systems, a different kind of stream is required.

Definition 3.6 (Dense Time Domain [Bro12])**.** *TD* is a dense time domain iff $TD \subseteq \mathbb{R}_+$ and

$$\forall x, y \in TD.x < y \implies \exists z \in TD.x < z < y$$

Definition 3.7 (Dense Timed Stream [Bro12])**.** A dense timed stream over a message domain M and a time domain $TD \subseteq \mathbb{R}_+$ is the total function $s : TD \to M$.

Cauchy continuity is used to ensure that small variations in the time lead to small variation in the message. This requires that the messages are a metric space.

Definition 3.8 (Cauchy Continuous Dense Timed Stream [Bro12])**.** A dense timed stream $s : TD \to M$ is *cauchy continuous* iff M is a metric space with distance function d and

$$\forall \epsilon \in \mathbb{R}_{>0}.\exists \delta \in \mathbb{R}_{>0}.\forall x, x' \in TD.|x - x'| < \delta \implies d(s(x), s(x')) < \epsilon$$

Hybrid State Machines [Hen96, LSVW96] can define behavior over dense streams. Furthermore, the analog stream can be discretized to an event stream. While information is lost during discretization, for many applications the digital approximation of the analog signals is sufficient. Different kinds of discrete, dense and super dense streams are listed in [RR11, SRS99].

Other Isabelle Streams

[GR06] also introduces a datatype for general streams and defines functions over these streams. Our definition of streams is an optimization of this definition for the newer HOLCF'11 [Huf12].

An alternative formalization of Focus in Isabelle is given in [Spi07]. This formalization uses Isabelle/HOL instead of Isabelle/HOLCF. Hence, it heavily relies on the finite `list` datatype. The stream datatype is defined using four constructors. One for finite timed streams (`FinT`), one for finite untimed streams (`FinU`), one for infinite timed streams (`InfT`), and one for infinite untimed streams (`InfU`):

```
datatype 'a streamSpichkova = FinT "'a list list" |
                              FinU "'a list"  |
                              InfT "nat ⇒ 'a list" |
                              InfU "nat ⇒ 'a"
```

This definition of streams has the downside that the kind of timing is not visible from the signature of a function. Thus, the type-checker cannot prevent the user from concatenating an untimed stream with a timed stream. Furthermore, having four constructors leads to many case-distinctions in definitions and lemmata. Using datatypes like `event` our general stream datatype can encode timing information without changing the definition of the general stream. This was demonstrated with event, time synchronous, and tone streams. Changing the stream-datatype for each kind of timing would bring additional overhead. Lastly, the HOLCF **`domain`** keyword brings many helpful properties like the prefix-order and induction out of the box. Especially for induction the cpo properties of our streams are helpful. With the admissibility it is possible show that a property holds on an infinite stream using induction.

Ptolemy II - Lee

Ptolemy II [Cla14] is a modeling and simulation framework for system design developed by UC Berkeley. The code is available under an open-source license. The frameworks include a graphical modeling tool named "Vergil". It is similar to other Component & Connector modeling languages like SysML [JPR+22] or UML [EBF+98, CKM+99, KER99, BC11, Rum16]. But Ptolemy focuses on the semantics of the models, the rendering of the visualization is only "incidental" [Cla14] as there may be multiple different visualizations.

Ptolemy is using *superdense* time. Every time information is encoded as $(t, n) \in (\mathbb{R} \times \mathbb{N})$. The *macrostep* t denotes the time of the event. Since it is a real number, continuous processes can be described. The microstep n is used to create a well-defined sequential order of "simultaneous" events. An example where micro steps are used are inelastic collisions such as in Newtons Cradle. The force is transmitted instantaneously through multiple spheres. A detailed explanation is given in [Lee17].

Microsteps are progress without the passing of time. Hence, it is possible to run into Zeno conditions, where only microsteps are increased, but no macrostep occurs. It is hard to statically analyze a model, whether it is zeno or not. [Lee17] shows a system which is zeno if the Collatz conjecture is false. Also seemingly simple systems can be zeno, for example a ball bouncing on the ground [Lee17]. Mathematically, a observation is encoded as $p : \mathbb{R} \times \mathbb{N} \to \mathbb{S}$ where $\mathbb{S}$ is the set of all possible signals. However, Ptolemy is using a different internal implementation of time. The macrostep is split up into two parts: One global floating-point constant *resolution* (r) and a natural number for the time slice (m). The time slice is saved as a Java `BigInteger` and is not susceptible to overflows. The time (t) is calculated from these values as $t = m * r$. An arbitrary large microstep would only lead to zeno-conditions, hence the microstep is encoded as a 32-bit integers. In most cases a overflow is caused by a zeno-condition.

With this implementation, the practical requirements from [BGL+15] are fulfilled. The precision can be set according to the requirements of the system. For example a high-speed processor could be simulated with a different precision than an clock. For continuous systems, the precision must be sufficiently fine to capture all relevant events (e.g. using Nyquist–Shannon sampling [Sha49]).

Ptolemy encourages the developer to create deterministic models. If a pseudo random number is required (e.g. for a Monte Carlo experiment [PCB+01]) the seed is managed by Ptolemy to achieve reproducibility. Since temporal semantics are an important part of Ptolemy, the timing properties are also handled deterministically. In a simulation this is relatively easy to achieve, since a global clock exists and one can "freeze" time if necessary. For an implementation in a real-life distributed system, the temporal properties are challenging to realize, as current hardware does not provide deterministic timing behavior. The PTIDES Model is addressing this issue [DFL+08].

Since Ptolemy models have a clearly defined semantic, it is possible to formally verify the models. In [CFL08] a transformation into Kripke structures is presented. [BÖFT09] used real-time maude [ÖM07] to verify liveness properties of Ptolemy models. Verification is also integrated into the visual editor.

Chapter 4

Timed Stream Bundles

Components in a distributed system communicate with other components via directed channels. Communication may occur on multiple input and output channels. Bundling the input or output streams together is therefore necessary. The concept of stream bundle (SB) is introduced as a function mapping channels to streams. General bundles extended to timed stream bundle (TSB) that are represented by a function mapping channels to timed streams. This chapter introduces an implementation of SBs in Isabelle as an extension of the implementation in [BKR+20]. The new implementation allows different timing modes and additional functions, allowing reuse of types, definitions, and theorems across timing modes.

4.1 Datatype Definition

This section presents the required definitions for introducing timed stream bundles (TSBs). Formally, an SB is a function that maps channels to streams, i.e., it associates channels with their communication history. Depending on the channel, its communication history can contain time information. To allow different abstract time information in communication histories, the `combinedStream` type ($M^{\underline{\cup}\omega}$) introduced in Section 3.2.6 is used.

Definition 4.1 (Stream bundle (SB))**.** Let C be a set of channel names, T_c the time-type, M_c the set of allowed messages for a channel $c \in C$, and $M = \bigcup_{c \in C} M_c$. The stream bundle type is then defined as:

$$SB_C \coloneqq \{sb \in (C \to M^{\underline{\cup}\omega}) \mid \forall c \in C.\ combValues(sb(c)) \subseteq M_c \\ \wedge combGetTime(sb(c)) = T_c\}$$

with

$$T_c = event \mid tsyn \mid tone \mid tsliced \mid untimed$$

A TSB is then simply a subtype of SB, where all channel histories are timed, i.e., $\forall c.$ `combGetTime` $(sb(c)) \neq untimed$ holds additionally. Depicting complete timed

communication histories over an infinite sequence of time intervals always results in infinitely long timed streams with infinite time progression on all channels. The time-mode of a stream in a bundle must be equal to its channel's time-mode. An untimed stream on a timed channel is not allowed. The correct time-mode is enforced by $combGetTime(sb(c)) = T_c$. An example of an correct bundle with different time-modes for its channels is shown in Fig. 4.1. On the first input channel of the TSB the channel's history is represented by a `bool event stream`. The second channel's history is depicted as a `nat tsyn stream`. In the following, C^{Ω} is the abbreviation for SBs, $C^{\underline{\Omega}}$ for TSBs, and $C^{*\underline{\Omega}}$ for bundles containing only event streams corresponding to channel set C.

Figure 4.1: Component with two input streams of different timing and message types

After this short theoretical introduction, the implementation in Isabelle is presented next.

Message Definition

To define SBs, channels and message types are introduced first. A datatype defining a superset of allowed messages exists similar to the untimed case:

```
datatype msg = NAT "nat" | BOOL "bool" | STRING "string" |
               INT "int" | RAT "rat"
```

Custom datatypes can be used by adding them to the `msg` type in Isabelle. This can be done manually, or by generating. This is especially relevant for message types that are defined, e.g., in Architecture Description Languages (ADLs) like MontiArc or SysML [KMP+21].

Timed Channel Definition

The `timeType` datatype (see Section 3.2.6) is necessary to support one the one hand TSBs where all channels have the same specific time-type, e.g., an event TSB or a sync TSB, and on the other hand TSBs where channels have different time-types as shown in Fig. 4.1.

The global datatype `msg` defines every message on every channel of a distributed system. Each channel restricts the possible messages to a subset of `msg`. As a result, the Isabelle encoding of channels is defined not only by a name but also by a `timeType` and a set of messages from `msg`.

```
datatype channel = CH (* name      : *) string
                      (* time type: *) timeType
                      (* messages : *) "msg set"
```

Simple functions for obtaining the time type or the allowed pure messages of a channel are defined:

```
fun cTime :: "channel ⇒ timeType" where
"cTime (CH _ tt _) = tt"
```

The function `cMsg` returns the set of all permitted messages of the given channel. It is independent from the time-type of the channel.

```
fun cMsg :: "channel ⇒ msg set" where
"cMsg (CH _ _ S) = S"
```

The domain of an SB is subject to a few assumptions. First and foremost, the domain must constitute a subset of the global channel type `channel`. A Representation (Rep) and Abstraction (Abs) function between the domain and the global channel type must be given, similar to the definition of a subtype in Isabelle. The Representation maps a domain element to the corresponding channel in the global channel type. The Abstraction function maps a global channel to its corresponding domain element, if it exists. Additionally, it must hold, that `Abs (Rep c) = c` holds. Additionally, the challenge of representing empty domains for SBs without any channels must be dealt with. Since Isabelle does not support empty types, the global channel type contains "empty" channels (`cEmpty` is the set of all empty channels). This allows defining empty domains with necessarily non-empty Isabelle types. Mapping some elements in a domain to an empty channel and others to non-empty channels is prohibited. The `chan` class defines an interface channel types using the assumptions discussed above. Furthermore, the channel set of a domain type `chanDom` is defined as all non-empty channels.:

```
class chan =
  fixes Rep :: "'a::type ⇒ channel"
  fixes Abs :: "channel ⇒ 'a"
  assumes abs_rep_id:"Abs (Rep a) = a"
  assumes chan_well:
      "range Rep ⊆ cEmpty ∨
       range Rep ∩ cEmpty = {}"
begin
  definition chanDom "'a itself => channel set" where
  "chanDom a = range Rep - cEmpty"
  ...
end
```

Through the mappings `Rep` and `Abs`, a subtype relation to the channel datatype is created. Together with the `abs_rep_id` assumption, each element of the type incarnates exactly one channel of the `channel` type. The second assumption `chan_well`

forces the type to represent either only existing channels or only contain channels in `cEmpty`. Furthermore, the domain of the type that only contains existing channels is defined as the range of `Rep` without any empty channels. Alternatively, defining the `Rep` function as an optional function removes the need for the second class assumption and makes `cEmpty` superfluous. However, it introduces additional checks whether `Rep` is a function or `None`. The intrinsic problem that types must be non-empty remains nonetheless.

Further subclasses of `chan` with additional assumptions for different time abstractions and timed channels, in general, are introduced hereafter.

Timed Classes for sets of channels are defined as follows. If a set of channels contains

- no untimed channel, it is in the class `tchan`.
- only untimed channel, it is in the class `uchan`.
- only timed event channels, it is in the class `echan`.
- only timed synchronous or time slice channels, it is in the class `tschan`.
- only timed channels with one message per time slice, it is in the class `tonechan`.
- only channels allowing multiple messages per time slice, it is in the class `mult_msg_chan`.
- only timed channels of the same timing mode, it is in the class `one_chan`.

Timed Bundle Definition in Isabelle

TSBs are a sub-domain of functions mapping channels (`chan`) to streams. The `sb_well` predicate restricts the messages on the streams to the allowed, i.e., correctly typed, messages of their corresponding channel using `cMsg`. Additionally, the time mode of the channel (`cTime`) must align with the combined stream type (`combGetTime`). Finally, to also allow definitions of functions without input or output channels, empty bundles must be supported. To define empty bundles as $\bot$, the necessarily existing element in the channel type must always map to the corresponding least stream (`combLeast`).

```
definition sb_well::"('c::chan ⇒ msg combinedstream) ⇒ bool" where
"sb_well f ≡ (∀c. combValues·(f c) ⊆ cMsg (Rep c))
           ∧ (∀c. combGetTime (f c) = (cTime (Rep c)))
           ∧ (∀c. cMsg (Rep c) = {} ⟶ (f c) = combLeast c)"
```

With this predicate, the SB type can be defined as the set of all functions from channels to streams that fulfill `sb_well`.

```
cpodef 'c::chan sb
     = "{f::('c::chan ⇒ msg combinedstream). sb_well f}"
```

Since the `combinedstream` type allows time-denoting messages for timed channels, the `sb` type in Isabelle also contains all TSBs. Additional types for timed and untimed bundles are not necessary.

When lifting the prefix order over combined streams to SBs, the bundle type automatically forms a pcpo. Different least bundles are not possible since `sb_well` allows exactly one empty stream per channel. Which empty stream is allowed depending on the channel's `timeType`.

In conclusion, a bundle type representing timed communication input or output histories was defined. This opens the door for the specification of timed components in distributed systems. Decoupling time abstractions and the bundle type definition enables extending the framework with additional time-modes. In later sections, the main focus lies in the general time abstraction using the event type, which is powerful enough to include the other introduced timing modes. A way to switch bundles from other time-types to event-time is given next. An overview for general timed and untimed bundle processing functions is given in Table 4.1. These functions ease defining components or systems presented in Chapter 5.

4.2 Changing Time Modes

Composing components with different time abstractions is not straightforward. Not only the channel's name but also the `timeType` must match. Hence, timed specifications and even different time abstractions lead to the necessity of converting bundles to other time abstractions. It allows abstraction changes in distributed systems and specifications. For this, time converter functions over bundles are implemented in Isabelle. The time converter functions allow mapping any bundle to an event-timed bundle and any bundle to an untimed bundle, i.e., they remove all timing information from a bundle. Using only these two converters, a user can define components of a timed system on different time abstraction levels and still compose them in the end without any loss of timing information. Furthermore, a time-independent system or component defined in a timed setting can be (re)used in an entirely untimed system and still behave correctly.

Conversion to Event Bundles

First, a function to change any channel's `timeType` is defined. This is used for converting the channels of bundles later on.

```
fun changeTimeType :: "timeType ⇒ channel ⇒ channel" where
"changeTimeType newTT (CH name _ P) = CH name newTT P"
```

	Signature	Description	See
	Abs_sb: $(cs \Rightarrow msg^{\underline{\cup\omega}}) \Rightarrow cs^{\Omega}$	lift a function to a SB	61
	Rep_sb: $cs^{\Omega} \Rightarrow (cs \Rightarrow msg^{\underline{\cup\omega}})$	represents SB as function	61
$\bot$	**bottom**: cs^{Ω}	least stream bundle	[BKR+20]
▶	**sbGetCh**: $cs^{\Omega} \Rightarrow cs^{\Omega} \rightarrow msg^{\underline{\cup\omega}}$	get the stream on channel	[BKR+20]
$^\frown_\Omega$	**sbConc**: $cs^{\Omega} \Rightarrow cs^{\Omega} \rightarrow cs^{\Omega}$	concatenation of bundles	[BKR+20]
	sbDrop: $\mathbb{N} \Rightarrow cs^{\Omega} \rightarrow cs^{\Omega}$	drops the first n elements	[BKR+20]
	sbTake: $\mathbb{N} \Rightarrow cs^{\Omega} \rightarrow cs^{\Omega}$	takes the first n elements	[BKR+20]
$\star$	**sbTypeCast**: $cs^{\Omega} \rightarrow ds^{\Omega}$	type conversion	[BKR+20]
$\uplus$	**sbUnion**: $cs^{\Omega} \rightarrow ds^{\Omega} \rightarrow (cs \cup ds)^{\Omega}$	merges two SBs together	[BKR+20]
	sbTakeWhile: $(msg \Rightarrow \mathbb{B}) \Rightarrow cs^{\Omega} \rightarrow cs^{\Omega}$	prefix while predicate holds	[BKR+20]
	sbDropWhile: $(msg \Rightarrow \mathbb{B}) \Rightarrow cs^{\Omega} \rightarrow cs^{\Omega}$	suffix while predicate holds	[BKR+20]
	sbNTimes: $\mathbb{N} \Rightarrow cs^{\Omega} \Rightarrow cs^{\Omega}$	iterate each stream n times	[BKR+20]
#	**sbLen**: $cs^{\Omega} \rightarrow \mathbb{N}_{\infty}$	length of the SB	[BKR+20]
$\#_{\surd}$	**sbTLen**: $cs^{\underline{\Omega}} \rightarrow \mathbb{N}_{\infty}$	complete time intervals	64
	sbTick: $cs^{\underline{\Omega}}$	one time interval	67
	tsbDrop: $\mathbb{N} \Rightarrow cs^{\underline{\Omega}} \rightarrow cs^{\underline{\Omega}}$	cut off first n time slices	66
	tsbRt: $cs^{\underline{\Omega}} \rightarrow cs^{\underline{\Omega}}$	cut off first time slice	66
	tsbTake: $\mathbb{N} \Rightarrow cs^{\underline{\Omega}} \rightarrow cs^{\underline{\Omega}}$	take n first times slices	66
	convUntimed: $cs^{\underline{\Omega}} \rightarrow cs^{\Omega}$	convert TSB to untimed SB	63
	tsbNth: $\mathbb{N} \Rightarrow cs^{\underline{\Omega}} \rightarrow cs^{\Omega}$	nth time slice	66
	tsbDelay: $\mathbb{N} \Rightarrow cs^{\underline{\Omega}} \rightarrow cs^{\underline{\Omega}}$	delay TSB by n time slices	67
	convEvent: $cs^{\underline{\Omega}} \rightarrow cs^{*\underline{\Omega}}$	convert TSB to event TSB	63
	convTimed: $cs^{\Omega} \rightarrow cs^{*\underline{\Omega}}$	convert untimed SB to an event TSB	63

Table 4.1: Definitions on TSB. Type-abbreviations are explained in Table 3.1

For changing the time type of the whole channel set `'cs` to the event type, the constructor `TLIVED` is defined.

```
datatype 'cs::one_chan tlived = TLIVED "'cs"
```

To be usable in the context of bundles, the `tlived` type has to be in the `chan` class. Representation, abstraction, and the channel domain are defined to correctly map `tstimed` channels to corresponding channels from the global channel datatype using `changeTimeType`. But not only the `timeType` of channels can change when converting a bundle. The messages of corresponding streams in the bundle have to change their time mode as well. A generalized function mapping messages from `msg combinedstream` to `msg event stream` is defined.

```
definition toEventS :: "msg combinedstream → msg event stream"
   where
"toEventS = combStreamCases ID tsyn2event tone2event
   tsliced2event undefined"
```

As a result, the definition of a time converter mapping for bundles of different time modes to an event-timed bundle is straightforward. The original stream is obtained and mapped to the event representation using `toEventS` for all channels.

```
definition convEvent::"'cs::{tchan} sb → 'cs tlived sb" where
"convEvent ≡ Λ sb. Abs_sb(λ (TLIVED c).
                                    TEVENT (toEventS·(sb ▶ c)))"
```

Conversion to Untimed Bundles

Similarly to converting to event bundles and streams, a datatype representing the untimed channels set is introduced. This can be helpful when checking an untimed property of a timed system because, in such cases, timing information is irrelevant and can be abstracted over.

```
datatype 'cs::one_chan untimed = UNT "'cs"
```

Again, not only channels but also messages have to be converted. For this, a function is defined to convert timed streams to an untimed `msg stream`.

```
definition toUntimedS :: "msg combinedstream → msg stream" where
"toUntimedS = combStreamCases e2ustream tsyn2ustream tone2ustream
   tsliced2ustream ID"
```

Then, using the introduced function, a converter removing all timing information from a bundle is introduced employing the same pattern as for `convEvent`.

```
definition convUntimed::"'cs sb → 'cs untimed sb" where
"convUntimed ≡ Λ sb. Abs_sb(λ(UNT c).
                                  UNTIMED( toUntimedS·(sb ▶ c)))"
```

Conversion to Timed Bundles

Adding time to an untimed SB is underspecified because timing information is created from nothing. Hence, the converter is represented by a set of functions to capture all possible timing information.

```
definition convTimed::"('cs::uchan sb → 'cs tlived sb) set" where
"convTimed ≡ {f | ∀ sb c. convUntimed (f sb) ▶ c = sb ▶ c}
```

The converter functions map untimed SBs to an event bundle, maintaining the messages on each channel. Removing time from the output bundle leads to the input bundle, i.e., `convTimed` contains inverse functions of `convUntimed`.

Conversion Functions for Alternative Time Representations

It is possible to convert to other timed SBs. Different time converter functions can be defined following the patterns of the event and untimed converters. But since the event streams are capable of representing more time information than other time representations introduced, there might be a loss of (timing) information. For example, consider a converter from event bundles to time slice bundles. Assume we wanted to keep the timing information. We thus chose the messages of the time slice stream to be lists. This enables us to effectively capture finitely many events in a single time slice. This conversion requires dropping unfinished time slices occurring in the event bundle. This is because it is impossible to represent unfinished time slices (unfinished lists) in a time slice stream.

4.3 Important Bundle Datatype Functions

A list of important functions over SBs and TSBs is given in Table 4.1. It extends previously introduced functions in [BKR+20] by additional operators developed specifically for timed bundles.

Functions over Timed Stream Bundles

The following sections introduce key TSBs functions.

Timed Length

The timed length of a TSB, i.e., the length of time represented by a TSB, is equivalent to the smallest number of time intervals of any stream of the TSB. The bundle without any channel is defined to have infinitely many time slices. The reasoning behind this is the causal behavior of components and explained in Section 5.1.1. Since the timed length of TSBs assumes only timed channels, this undefined case never occurs. In the following definition, the `LEAST` operator obtains the smallest number from a set containing timed length for each stream of the TSB.

```
definition sbTLen :: "'cs::tchan^Ω ⇒ lnat" where
"sbTLen sb ≡ if chDomEmpty TYPE('cs) then ∞
                  else LEAST n. n∈{#√(sb ▶ c) | c. True}"
```

For time-synchronous bundles, the timed length is equivalent to the regular length. The continuity of `sbTLen` behaves analogously to `sbLen`, i.e. the function is only continuous for finitely many channels.

```
lemma sbtlen_cont: "cont (sbTLen::'cs::{finite,tchan}^Ω ⇒ lnat)"
```

The restriction to finitely many channels is necessary because infinite chains exist over infinitely many channels, which contradicts the continuity property. Consider the following chain of bundles with infinitely many channels. The first and lowest element has an empty stream on every channel. Its timed length is 0. The chain continues by adding a time slice to one channel in each step. Since there are infinitely many channels, an empty stream on some channel still exists for each chain element. The following illustration shows the first three elements of the chain:

$$\bot \sqsubseteq [c_1 \mapsto \langle\surd\rangle, c_2 \mapsto \varepsilon, \dots] \sqsubseteq [c_1 \mapsto \langle\surd\rangle, c_2 \mapsto \langle\surd\rangle, c_3 \mapsto \varepsilon, \dots] \sqsubseteq \dots$$

The timed length `sbTlen` of each chain element is 0, and so is the least upper bound of the chain of timed lengths. However, the least upper bound of the chain of TSBs contains one time slice on every channel; the `sbTLen` function of this least upper bound evaluates to 1. Thus, a continuous `sbTLen` function for TSBs with infinitely many channels is impossible.

Mapping timed stream functions

One crucial definition over TSBs is a function that maps existing functions over timed streams, e.g., `tsdrop` from Table 3.4 to a bundle's streams. For this, we first introduce `map_combstream` which simply applies the correct function to the combined stream

```
fun map_combstream::"('m event stream → 'm event stream)
  ⇒ ('m tsyn stream → 'm tsyn stream)
  ⇒ ('m tone stream → 'm tone stream)
  ⇒ ('m list stream → 'm list stream)
  ⇒ ('m stream → 'm stream)
  ⇒ 'm combinedstream ⇒ 'm combinedstream" where
"map_combstream f _ _ _ _ (TEVENT s) = TEVENT (f·s)"    |
"map_combstream _ f _ _ _ (TSYN s) = TSYN (f·s)"        |
"map_combstream _ _ f _ _ (TONE s) = TONE (f·s)"        |
"map_combstream _ _ _ f _ (TSLICED s) = TSLICED (f·s)"  |
"map_combstream _ _ _ _ f (UNTIMED s) =  UNTIMED (f·s)"
```

This is then lifted to the bundle level.

```
definition tsbMap::"(msg event stream → msg event stream)
  ⇒ (msg tsyn stream → msg tsyn stream)
  ⇒ (msg tone stream → msg tone stream)
  ⇒ (msg list stream → msg list stream)
  ⇒ 'cs::tchan sb → 'cs sb" where
"tsbMap f1 f2 f3 f4 = ( Λ sb. Abs_sb (λc. map_combstream f1 f2 f3
    f4 undefined ((Rep_sb sb) c)))"
```

Dropping time slices

Time slices at the start of a TSB can be dropped by applying the `tsdrop` operator to each stream on every channel. The first parameter is a natural number indicating how many time slices shall be dropped.

```
lift_definition tsbDrop::"nat ⇒ 'cs::tchan^Ω → 'cs^Ω" is
"tsbDrop n = tsbMap (tsdrop n) (tsdrop n) (tsdrop n) (tsdrop n)"
```

The `tsbRt` function is an abbreviation for `tsbDrop 1`, i.e., it drops only the first time slice.

Taking time slices

Time slices at the start of a TSB can be taken by applying the `tstake` operator to each stream on every channel. All subsequent time slices are dropped. A natural number is given to the `tsbTake` function as a parameter to define how many time slices shall be taken.

```
lift_definition tsbTake::"nat ⇒ 'cs::tchan^Ω → 'cs^Ω" is
"tsbTake n = tsbMap (tstake n) (tstake n) (tstake n) (tstake n)"
```

The `tsbHd` function is an abbreviation for `tsbTake 1`, i.e., taking only the first time slice and dropping the rest. The nth time slice can be obtained by applying the `tsnth` function to each stream on every channel. After obtaining the time slice, no additional timing information remains in the result. The first parameter is a natural number indicating which time slices shall be taken.

```
definition tsbNth :: "nat ⇒ 'cs::{tchan}^Ω → 'cs^Ω" where
"tsbDrop n = tsbMap (tsnth n) (tsnth n) (tsnth n) (tsnth n)"
```

Introducing Delay

A delay is introduced to a TSB by adding a time slice in front of each stream on every channel. Delaying an empty bundle, i.e., the bundle with an empty stream on all its channels, results in `tsbTick`. This bundle contains one empty time slice on all its streams for all time modes but `TOne`. A message must be chosen as `TOne` cannot have empty time slices.

The helper function `OneTLen` produces an empty, or, in the case of `TOne`, random time slice. Its first parameter indicates the desired timing. The second parameter is only for `TOne` type and contains the set of possible messages to choose from.

```
fun OneTLen::"timeType ⇒ msg set ⇒ msg combinedstream" where
"OneTLen TTOne msgs = TONE (↑(TOneMsg (SOME x. x∈msgs)))" |
"OneTLen TTFin  _   = TSLICED (↑[])" |
"OneTLen TTLive _   = TEVENT (↑Tick)"|
"OneTLen TTsyn  _   = TSYN (↑Eps)"
```

Next, we employ the helper to define `tsbTick`, a TSB with exactly one time slice on every channel. On a per-channel basis, the allowed messages and timing are retrieved using `cMsg` and `cTime`. Then `OneTLen` produces an empty or random time slice.

```
definition tsbTick :: "'cs::tchan^Ω" where
"tsbTick = (if (chDom TYPE('cs) = {}) then ⊥
     else Abs_sb (λc. OneTLen (cTime (Rep c)) (cMsg (Rep c))))"
```

The `tsbDelay` function adds delay in front of a TSB. For this, the `sbTick` bundle is reused. It is delayed by appending the `sbTick` bundle in front of a TSB. To define the delay's length, a natural number is given to the `tsbDelay` function.

```
definition tsbDelay :: "nat ⇒ 'cs::{tchan}^Ω → 'cs^Ω" where
"tsbDelay n ≡ sbConc (sbNTimes n tsbTick)"
```

4.4 Wellformedness

Like timed streams, a TSB is not necessarily wellformed. An event stream in a TSB may contain infinitely many messages but not infinitely many ticks. Thus, we define wellformed bundles as bundles that contain only wellformed combined streams.

```
definition wellformed_sb::"'a::tchan sb ⇒ bool" where
"wellformed_sb sb = (∀c. wellformed (sb ▶ c))"
```

At first glance, an alternative approach is to define the wellformedness of TSBs analogously to that of streams. Thus, a TSB would be wellformed, if $\#sb = \infty \longrightarrow \#_{\surd} sb = \infty$ holds. But by looking at the following "wellformed" bundle

$$sb = [c1 \mapsto \langle 1 \rangle^{\infty}, c2 \mapsto \epsilon]$$

it becomes apparent that this alternative definition includes undesired bundles where not all infinite streams necessarily have infinitely many Ticks. This behavior results from the length definition for stream bundles, which always obtains the length of the shortest stream. Redefining the length to the maximum leads to analogous problems. As a result, this approach does not specify the desired wellformedness.

4.5 Characterization of Bundles as Streams

Any TSB with finitely many channels and the same number of time slices can be represented as an event stream of channel and message pairs. The resulting sequential representation is especially useful for processing each event individually. For stream bundles with infinitely many channels or varying numbers of time slices, the stream can only represent a prefix of the TSB.

Example 4.1 (Stream defining a TSB). *The stream* $\langle (c_1, 5), \surd, (c_2, True), (c_1, 9), \surd \rangle$ *over channel domain* $\{c_1, c_2\}$ *defines the bundle* $[c_1 \mapsto \langle 5, \surd, 9, \surd \rangle,\ c_2 \mapsto \langle \surd, True, \surd \rangle]$.

However, these streams must only contain allowed pairs where the messages are allowed to flow on the accompanying channel. The following predicate `well` holds for such pairs. While the predicate `well` defines a wellformedness property for stream messages, the `wellformed` predicate defines a property over the stream's timing information.

```
definition well::"('chan::chan × msg) ⇒ bool" where
"well ≡ λ(c,m). (m ∈ cMsg (Rep c))"
```

A new type containing only the `well` message and channel pairs is defined. The "well" pair is undefined for channels without any allowed messages. This is a result of the inability to define empty types in Isabelle.

```
typedef ('chan::chan) msgWithChan = "{(chan, msg) | (chan::'chan)
   msg::msg. well (chan,msg) }
    ∪ (if (chDom TYPE('chan) = {}) then {undefined} else {})"
```

Each time slice of a `'cs msgWithChan event stream` contains all messages of each channel from the corresponding time slice. The result matches the channel's history by filtering the stream only to contain messages of a specific channel and Ticks with `eventFilter`.

```
definition eventFilter:: "'cs ⇒ 'cs::{countable,chan} msgWithChan
   event stream → msg event stream" where
"eventFilter c = (emap snd) oo (efilter (λ(ch,m). ch=c))
                  oo (emap Rep_msgWithChan)"
```

Using `eventFilter` over all channels to form a SB is done by the `uSplit` function. Additionally, it can convert the channel's history from `event stream` to another time mode, with possible information loss. A helper function mapping `msg event stream` to `msg combinedstream` is defined for this.

```
fun toMsg::"timeType ⇒ msg event stream → msg combinedstream"where
"toMsg TUntimed  = (Abs_cfun UNTIMED) oo e2ustream"|
"toMsg TTFin  = (Abs_cfun TSLICED) oo sEventToList"|
"toMsg TTLive = Abs_cfun TEVENT" |
"toMsg _ = undefined"
```

Maintaining the event timing is recommended to avoid information losses. By restricting the bundle only to contain untimed, event, or time slice streams, the undefined case in `toMsg` never occurs in `uSplit`.

```
lift_definition uSplit::"'cs::{countable,mult_msg_chan}
   msgWithChan event stream → 'cs sb" is
"λ s. if chDom TYPE ('cs) = {}
                  then ⊥
                  else Abs_sb(λc. toMsg (cTime (Rep
                     c)) · (eventFilter c·s))"
```

This splitting operator is surjective for wellformed SBs with finitely many channels and the same number of $\surd$ on each channel. The limitation to finitely many channels is required to prevent infinitely long time slices, as demonstrated by the following example. Assume an SB with infinitely many channels indexed by the natural numbers. All channels map to the stream of infinitely many time slices containing a single message, 1 each:

$$(\lambda n.\langle 1, \surd\rangle^\infty)$$

Representing this TSB as a single event stream is impossible since the first time slice would contain infinitely many pairs $(1, n)$ without ever reaching the first $\surd$, thus contradicting the continuity of time. However, for SBs with finitely many channels, usually, multiple representations can be defined. Merging all channel histories into one stream is non-deterministic because the order between two messages of different channels in the same time slice is underspecified. Any permutation maintaining the internal order of each channel is acceptable. In conclusion, a merging operator is defined as a set of continuous functions whose output is a prefix of the input bundle. Equality is not possible since continuity and unfinished time slices lead to contradictions.

```
definition mergeGen :: "('cs::{mult_msg_chan,finite} sb → 'cs
    msgWithChan event stream) set" where
"mergeGen = {f | f. ∀sb. uSplit·(f·sb) ⊑ sb}"
```

Additional properties restrict the merge specification to mitigate the issue of the output being a prefix of the input. Each merge function should output at least all time slices of the bundle that are complete in all streams. When merging wellformed infinite timed streams, the merged stream contains all input elements from all input streams in the element's resprective timeslices [Bro88, KPR+25b]. Furthermore, in the case of the TSB without any channels, only time progression, i.e. the infinite stream of $\surd$s, is produced.

```
definition mergeTimed::"('cs::{tchan,finite,mult_msg_chan} sb →
    'cs msgWithChan event stream) set" where
"mergeTimed= mergeGen
  ∩ {f| f. if chDom TYPE ('cs) = {}
      then ∀sb. f·sb = ((↑Tick)^∞)
      else ∀sb. #√sb = #√(uSplit·(f·sb))}"
```

Timed merge functions provide a serialization that can be used as input for state-based components (e.g., automata) with iterative and element-wise processing of bundles while preserving timing information.

4.6 Discussion

In this section, a TSB datatype implementation in Isabelle was introduced as an extension of the previous implementation of SB from [BKR+20]. This implementation offers differently timed bundles while still supporting composition by adding the `timeType` to channels. Moreover, this allows TSB with differently timed channels in the same bundle and opens the door to composing components with divergent time abstractions. Moving the timing mode to the message and channel datatypes in the Isabelle implementation enables further time abstraction extensions.

Separate type definition for TSBs

Introducing a new type for TSBs hinders the re-usage of already defined functions and proven theorems. This can only be partially mitigated by offering additional lifting for functions and theorems. Additionally, without special case-by-case composition operators, it is type-wise not possible to compose functions that have untimed and timed parts, e.g., a parallel composition of `convTimed` as seen in Section 4.2 and `convUntimed` as seen in Section 4.2

Bundles with equally long streams

Restricting the presented SB type to equally long streams on all bundles regarding time slices is challenging. On the one hand, the merge function output would always contain the complete message histories of all channels, and the (timed) length functions would always be continuous. On the other hand, different channels of one bundle can not be delayed differently.

Example 4.2 (TSBs with equally long streams)**.** *Timed bundles with the same amount of time slices per channel, e.g.,* $[c_1 \mapsto \langle 5, 3, \surd, 9, \surd\rangle,\ c_2 \mapsto \langle \surd, True, \surd\rangle]$ *are TSBs with equally long streams. A bundle with differently many time slices, e.g.,* $[c_1 \mapsto \langle 5, 3, \surd\rangle,\ c_2 \mapsto \langle \surd, True, \surd\rangle]$ *is not allowed.*

Composing components with different delays generally introduces problems for equally long streams. Additionally, process iterations of an event-driven automaton would not necessarily produce an SBs.

Bundles with infinitely long streams

Limitation to only infinitely long histories on all channels fixes the problem of equally long streams TSBs, similar to [BR07].

Example 4.3 (TSBs with infinitely longstreams)**.** *Timed bundles with infinitely long streams per channel, e.g.,* $[c_1 \mapsto \langle 5, \surd \rangle^\infty,\ c_2 \mapsto \langle \surd, True, \surd \rangle^\infty]$ *are TSBs with infinitely long streams. A bundle containing a finite stream, e.g.,* $[c_1 \mapsto \langle 5, \surd \rangle,\ c_2 \mapsto \langle \surd, True, \surd \rangle^\infty]$ *is not allowed.*

However, it hinders iterative processing like fixed point calculations used in Section 5.1.4 for composition. Since no actual prefixes of bundles can be obtained, causality predicates can not be defined over the TSB type. Instead, a different type capable of representing such prefixes is needed, e.g., finite SBs. In general, a restriction of the TSB type would reduce the specification options of a user since recursion or iteration is no longer available.

Streams characterizing TSBs

In Section 4.5, a possibility to represent bundles as streams was introduced. However, using streams without ever introducing SBs leads to many problems. Firstly, streams can only represent equally long bundles and thus have the same restrictions. Furthermore, components transmitting messages to different target components in a system need filters in between to allow only messages from specific channels to go through.
Representing bundles by tuples of streams instead allows accurate capturing of all possible finite bundles. But again, the composition does not generally work. It would be necessary to define composition operators for all possible tuple sizes.

Bundles with generic message type

On another note, not supporting generic message types is a necessary drawback of allowing only a subset of messages on channels. A possible workaround is moving the theoretical concepts into a locale in Isabelle. Such parametric theories can be interpreted later by defining parameters and proving their assumptions. A message type and a mapping of channels to allowed messages could be such parameters. Nevertheless, since locales do not allow datatype definitions with only locally fixed parameters, e.g., the message type, this is not a viable option.

Time granularity

As introduced in Chapter 3, `Ticks` on different streams are equidistant. Thus, all nth `Ticks` occur simultaneously. This can lead to problems concerning the readability and writability of specifications when dealing with fast and slow-reacting components in the same system. Choosing a very small time slice length for $\surd$s to accommodate the fast-reacting component accurately leads to overcrowding the channel of the slow component with $\surd$s. On the other hand, using a very rough time granularity leads to the loss of information for the fast-reacting component as depicted in Fig. 3.2 when compared to

Fig. 3.3. Introducing a finer granularity from a rough granularity leads to uncertainty. It is not clear where (additional) Ticks should be placed.

Example 4.4 (TSBs with different time granularities)**.** *The bundle* $[c_1 \mapsto \langle 1, 1, 1, 1, 1, 1, 1, 1, \surd\rangle^\infty,\ c_2 \mapsto \langle \surd, \surd, \surd, \surd, True\rangle^\infty]$ *could be equivalent to* $[(c_1, 8) \mapsto \langle 1, \surd\rangle,\ (c_2, \frac{1}{4}) \mapsto \langle \surd, True\rangle^\infty]$ *with granularity factors of* 8 *for* c_1 *and* $\frac{1}{4}$ *for* c_2*.*

Extending our channels with a time granularity factor would allow conversion and comparison of channel histories and their messages.

Chapter 5

Timed Component Specifications

This chapter introduces deterministic and non-deterministic components. These components can be combined into large systems using *composition*. Non-deterministic components typically appear early in the design process, containing underspecification. Their behavior can be *refined* through the addition of constraints.

5.1 Deterministic Component Specifications

This section introduces deterministic components that map each input to exactly one output. Programs in general programming languages (e.g., Java) are often equivalent to deterministic functions.

5.1.1 Causality

Untimed deterministic components are defined as continuous functions. However, for timed deterministic components, additional restrictions are required. The following example demonstrates this with a continuous timed function, which is impossible to realize in practice.

Example 5.1 (Unrealizable Timed Function)**.** *The function* `edrop 1` *deletes the first time slice and returns all subsequent time slices unchanged. The following equation exemplifies the function* `edrop`*:*

$$edrop\ 1\ \langle Tick, EventMsg\ 3, Tick\rangle = \langle EventMsg\ 3, Tick\rangle.$$

This function is considered an oracle as the first time slice of the output depends on the second time slice of the input, allowing it to "look into the future".

To forbid such impossible behavior, *causality* is introduced. Weak causality ensures that the n-th time slice of the output depends only on the first n time slices of the input. Hence the n-th time slice of the output cannot depend on the n+1-th input time slice. Weakly causal components can act immediately and do not necessarily add delay. Additionally, strongly causal components always add delay. Strong causality ensures that the n+1-th time slice of the output depends only on the first n time slices of the input. Strong causality is thus a further restriction and a special case of general

(weak) causality. In Section 5.1.4 we will show the particular benefits of strongly causal functions.

To define causality, we only use the `tlen` ($\#_{\surd}$) function, which returns the number of time slices in a given bundle. Together with monotonicity this ensures the intended behavior.

```
definition weak::"('I::{cpo,tlen} → 'O::{cpo,tlen}) ⇒ bool" where
"weak spf ≡ ∀i. #√i ≤ #√(spf·i)"
```

The unrealizeable function from example 5.1 does not fulfill the weakness-predicate:

$$\#_{\surd}\langle Tick, EventMsg\ 3, Tick\rangle = 2 \not\leq 1 = \#_{\surd}\langle EventMsg\ 3, Tick\rangle$$

The definition of strong causality is similar to weak causality, only a '+1' is added on the left side of the equation to account for the delay:

```
definition strong::"('I::{cpo,tlen} → 'O::{cpo,tlen}) ⇒ bool"
    where
"strong spf ≡ ∀i. (#√i) + 1 ≤ #√(spf·i)"
```

It is not possible to define strong causality by replacing the $\leq$ in the definition of weak causality with $<$. The result would be $\#_{\surd}i < \#_{\surd}(spf\ i)$, and for input i with infinite time slices this constraint is inconsistent, i.e., cannot be met.

Every strongly causal function is also weakly causal:

```
lemma assumes "strong spf"
  shows "weak spf"
```

A special case are components without input channels. Since the empty bundle is interpreted as having infinitely many time slices, components with the empty bundle as input must produce an output with infinitely many time slices to fulfill the causality property. Components without output channels are always strongly causal.

Relation between Causality and Continuity

The previous definition of weak causality took a continuous function as input. This section proves that in certain cases, continuity follows from causality.

```
lemma weak_mono2cont: (* spf is over tsyn stream! *)
  fixes spf::"'in tsyn stream ⇒ 'out tsyn stream"
  assumes "⋀i. #√i ≤ #√(spf i)"
      and "monofun spf"
  shows "cont spf"
```

The proof uses the fact that for every infinite chain Y over `tsyn stream` the length of the least upper bound is infinite ($\#_{\surd}(Lub\ Y) = \infty$).

For `event stream` and bundles this property does not hold, as we can construct a counterexample `c_ex`:

```
definition c_ex::"unit event stream ⇒ unit event stream" where
"c_ex i ≡ if (i = ↑(EventMsg ())^∞) then ↑Tick
          else sfilter {Tick}·i"
```

The counterexample `c_ex` is a weakly causal and monotonic function:

```
lemma "#√i ≤ #√ (c_ex i)"
  and "monofun c_ex"
```

But `c_ex` is not a continuous function. Consider the infinite chain with only *EventMsg* messages. The least upper bound of this chain is the not-wellformed stream $(\mathit{EventMsg}\ ())^\infty$. All elements of the chain are mapped by `c_ex` to ϵ, but the least upper bound of the chain is mapped to `↑Tick`. Thus, the function `c_ex` is not continuous. Hypothetically, with a relaxed definition of continuity, which only considers wellformed least-upper-bounds, the proof would work similarly to the proof over `tsyn stream`.

Comparison to the Causality Definition by Broy

Our causality definition differs from the causality definition by Broy [BS01b]. In this section we will compare the definitions of weak causality and discuss the decisions. For strong causality the reasoning is analogous and hence omitted for brevity. The causal functions defined in this section are similar to *winning strategies* by [BS01b]. A winning strategy is a function τ which maps an input with infinite time slices to an output with infinite time slices: $\tau \in I^{\underline{\infty}} \to O^{\underline{\infty}}$. Additionally, the following causality constraint must hold:

$$\forall x, y \in I^{\underline{\infty}}; t \in \mathbb{N} : x \downarrow_t = y \downarrow_t \implies \tau(x) \downarrow_t = \tau(y) \downarrow_t$$

Here, $x \downarrow_t$ returns the first t time slices of x. Monotonicity is not defined, since $I^{\underline{\infty}}$ and $O^{\underline{\infty}}$ don't have an prefix order ($\sqsubseteq$). From every winning strategy, we can construct a weakly causal function.

$$\begin{aligned} &\mathit{toTSPF} :: (I^{\underline{\infty}} \to O^{\underline{\infty}}) \to I^{\underline{\Omega}} \to O^{\underline{\Omega}} \\ &\mathit{toTSPF}(\tau, x) := \tau(x^\frown \mathit{Tick}^\infty) \downarrow_{(\#_{\surd} x)} \end{aligned}$$

First, the potentially finite input $x \in I^{\underline{\Omega}}$ is concatenated with an infinite amount of *Tick* messages. The resulting input has infinite ticks[1] and is in $I^{\underline{\infty}}$. Then, τ is applied and produces an infinite output. This output is restricted to the same number of time slices as the original input x. The definition *toTSPF* always produces a weakly causal function. Since $I^{\underline{\infty}} \subseteq I^{\underline{\Omega}}$ and weak causality guarantees that infinite inputs produce an infinite output, converting from TSPF to Broy is simpler:

[1] We omit wellformedness for the proof sketch.

$$toBroy \in (I^{\underline{\Omega}} \to O^{\underline{\Omega}}) \to I^{\underline{\infty}} \to O^{\underline{\infty}}$$
$$toBroy(tspf, x) := tspf(x)$$

$I^{\underline{\Omega}} \to O^{\underline{\Omega}}$ contains more information than $I^{\underline{\infty}} \to O^{\underline{\infty}}$. Take for example:

$$tspf_w \in I^{\underline{\Omega}} \to O^{\underline{\Omega}} \qquad tspf_s \in I^{\underline{\Omega}} \to O^{\underline{\Omega}}$$
$$tspf_w(x) := Tick^{\#\surd x} \qquad tspf_s(x) := Tick^{\#\surd x+1}$$

$$toBroy(tspf_s) = toBroy(tspf_w) = \lambda\_.Tick^{\infty}$$
$$toTSPF(toBroy(tspf_s)) = tspf_w$$

Generally, the following properties hold:

$$\forall \tau \in I^{\underline{\infty}} \to O^{\underline{\infty}} : toBroy(toTSPF(\tau)) = \tau$$
$$\forall tspf \in I^{\underline{\Omega}} \to O^{\underline{\Omega}} : toTSPF(toBroy(tspf)) \sqsubseteq tspf$$

Causality and Realizability

While causality was introduced to remove unrealizeable behavior, there are still problematic corner cases regarding time and reaction to time. The following function demonstrates such a case:

$$spf \in \mathbb{N}^{*\underline{\omega}} \to \mathbb{N}^{*\underline{\omega}}$$
$$\forall ts.spf(EventMsg\ \_\hat{}ts) = spf(ts)$$
$$\forall ts.spf(Tick\hat{}ts) = \langle EventMsg\ 42, Tick\rangle\hat{}spf(ts)$$

All messages are ignored, only *Tick*s produce output. Every *Tick* in the input first leads to the output *EventMsg* 42 followed by a *Tick*. So even though the input time slice is complete (as marked by the input *Tick*), it is possible to send a message before the output time slice is complete. Either there is no global time, and the output time is slightly behind the input time. This would lead to problems in the feedback semantics. Or alternatively, the component "knows" before the time slice ends that no more input-messages will arrive. Both explanations are problematic. A clean solution are strongly causal functions, where this problem does not occur. When selecting a small enough time-interval all real components are strongly causal. For example, even an optical-wire has a certain delay due to the speed of light. For I/O automata [Rum96], it is possible to add conditions over the transition functions, which forbid the behavior described above.

5.1.2 Datatype for Deterministic Component Specifications

We define a timed stream processing function (TSPF) as a continuous and causal bundle-to-bundle function with fixed input and output channels [Rum96]. Monotonicity of the function implies that a component can not take back an already produced output. Continuity ensures that a component behaves the same on an infinite input as it would on its finite prefixes. Causality guarantees that the component cannot look into the future.

Definition 5.1 (timed stream processing functions (TSPFs) based on total functions)**.** Let C be the set of all possible timed channels and $I, O \subseteq C$. We can define the timed stream processing function (TSPF) type $TSPFw_{I,O}$ that includes all continuous and weakly causal TSPFs with input channels I and output channels O as shown below:

$$TSPFw_{I,O} := \{f \in I^{\underline{\Omega}} \to O^{\underline{\Omega}} \mid weak\ f\}$$

For strongly causal functions the type $TSPFs_{I,O}$ is defined as:

$$TSPFs_{I,O} := \{f \in I^{\underline{\Omega}} \to O^{\underline{\Omega}} \mid strong\ f\}$$

Isabelle Definition

Because Isabelle enforces non-emptiness of datatypes, we introduce an additional class. This class ensures the existence of an element with infinite ticks.

```
class tlen_inf = tlen +
  assumes tlen_inf: "∃a::'a. #√a = ∞"
begin
  definition tlen_inf::"'a" where
  "tlen_inf = (SOME a. #√a = ∞)"
end
```

This element with infinite ticks is used to create a constant, strongly causal function.

```
lemma strong_tleninf: "strong (Λ _ . tlen_inf)"
```

Hence, there exists a weakly causal function. The datatype `tspfw` is defined:

```
cpodef ('I::"{tlen,cpo}",'O::"{tlen_inf,cpo}") tspfw =
    "{f::('I → 'O) . weak f}"
```

Strongly causal functions are defined as a subset of weakly causal functions:

```
cpodef ('I::"{tlen,cpo}",'O::"{tlen_inf,cpo}") tspfs =
    "{f::('I,'O)tspfw . strong (Rep_tspfw f)}"
```

The datatypes are abbreviated as $\to_w$ and $\to_s$. For example, the strongly causal function between input bundle $I^{\underline{\Omega}}$ and output bundle $O^{\underline{\Omega}}$ is written as $I^{\underline{\Omega}} \to_s O^{\underline{\Omega}}$.

5.1.3 Wellformedness of Component Specifications

Proving wellformedness of inputs and outputs can be quite cumbersome and a burden on users. Thankfully, causality takes care of all wellformedness proofs. Every weakly causal function with wellformed input always produces a wellformed output.

```
lemma wellformed_weakI:
  fixes spf::"'csIn::wellformed → 'csOut::wellformed"
  assumes "weak spf" and "wellformed sb"
  shows "wellformed (spf·sb)"
```

5.1.4 Composition of Component Specifications

To model complex systems, the complexity can be spread across multiple subsystems. This process of divide and conquer is also called decomposition. Combining the subsystems to form the (complex) system is called composition. Composition uses channel names to connect output-ports to input ports of components. The result of a composition is again a TSPF. This allows to hierarchically create large networks able to precisely model complex behavior while the individual pieces remain easily understandable and verifiable.

```
fixrec spfComp::"('I1^Ω → 'O1^Ω) → ('I2^Ω → 'O2^Ω)
    → ((('I1 ∪ 'I2) - ('O1 ∪ 'O2))^Ω → ('O1 ∪ 'O2)^Ω)" where
"spfComp·spf1·spf2·sb = spf1·((sb ⩊_ spfComp·spf1·spf2·sb)⋆1)
                      ⩊ spf2·((sb ⩊_ spfComp·spf1·spf2·sb)⋆2)"
```

Assuming the output channels are disjoint, the composition is commutative and associative. This and specific modes of composition are discussed in [BKR+20]. In the following we will focus on timed properties.

Composition of Strongly Causal Components

Even in complex composition scenarios as shown in Fig. 5.1, the composition of strongly causal TSPF always results in a strongly causal TSPF:

```
lemma spfcomp_causal:
  assumes "strong f" and "strong g"
  shows "strong (f ⊗ g)"
```

We introduce a new definition to easily compose strongly causal components. The signature is similar to the untimed composition, but all components are strongly causal.

```
definition spfsComp::
"('I1::tchan^Ω →s 'O1::tchan^Ω) → ('I2::tchan^Ω →s 'O2::tchan^Ω)
    → ((('I1 ∪ 'I2) - ('O1 ∪ 'O2))^Ω →s ('O1 ∪ 'O2)^Ω)"
```

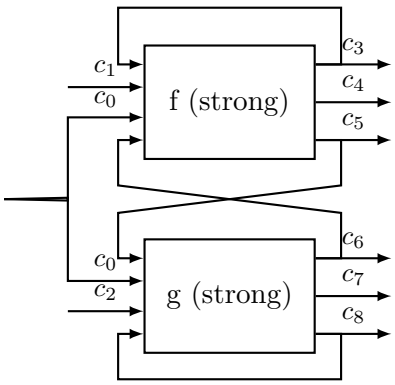

Figure 5.1: Complex composition scenario [BKR+20]

Composition of Weakly Causal Components

The composition of weakly causal functions is more challenging than composing strongly causal functions. Figure 5.2 shows the composition of two weakly causal components. For simplicity we use the `TOne` time, where each time slice contains exactly one message. The first component `ADD` takes two streams of natural numbers and adds them pairwise. The second component `SUC` takes a stream of natural numbers and increases each value by one.

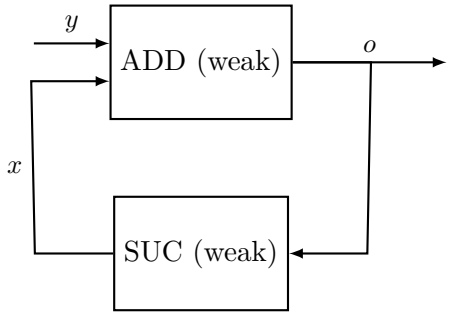

Figure 5.2: Composition of two weakly causal functions.
Resulting system is not causal.

The recursive equation of the composed system is:

$$\forall y \in \mathbb{N}^{\underline{1\omega}}.\exists x \in \mathbb{N}^{\underline{1\omega}}.\ x = map\ (\lambda(n,m).1+n+m)(zip\ x\ y)$$

For the first message x_0 the following equation must hold: $x_0 = 1+x_0+y_0$. Since $y_0 \in \mathbf{N}$ there exists no x_0 which fulfills the condition. Hence the only solution is $x = \bot$ and the composed behavior is not weakly causal. In contrast, sequential and parallel composition (Fig. 5.3) of weakly causal functions always produce a weakly causal function.

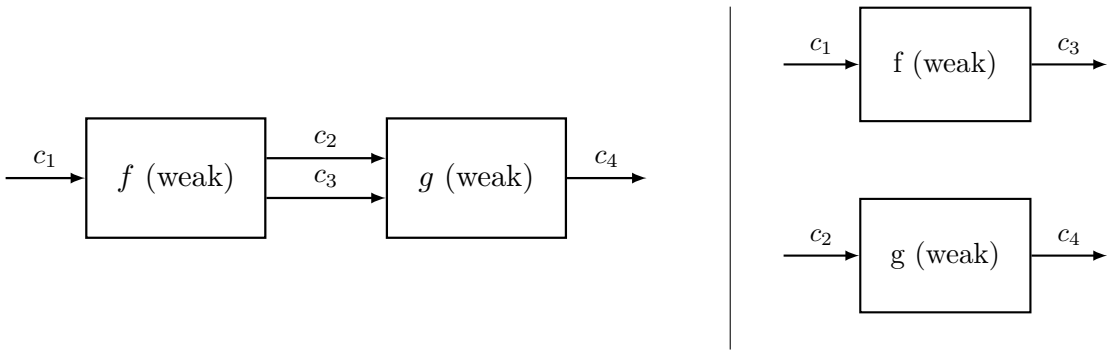

Figure 5.3: Sequential and parallel composition example [BKR+20]

This is proven in the following lemmata. The assumptions ensure that no feedback channel exists.

```
lemma spfcomp_nofeed_weak1:
  fixes f::"'fIn::tchan^Ω → 'fOut::tchan^Ω"
    and g::"'gIn::tchan^Ω → 'gOut::tchan^Ω"
      (* No self-loops: *)
  assumes "chDom TYPE('fOut) ∩ chDom TYPE('fIn) = {}"
      and "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
  (* No feedback between components, only sequential allowed: *)
    and "chDom TYPE('gOut) ∩ chDom TYPE('fIn) = {}"
    and "weak f" and "weak g"
  shows "weak (f ⊗ g)"
```

For weakly-causal parallel and sequential composition specialized definitions exist:

```
definition spfwCompPar::
  "('I::tchan^Ω →w 'O::tchan^Ω) → ('J::tchan^Ω →w 'P::tchan^Ω)
      → ('I ∪ 'J)^Ω →w ('O ∪ 'P)^Ω"

definition spfwCompSeq::
  "('I::tchan^Ω →w 'L::tchan^Ω) → ('L^Ω →w 'O::tchan^Ω)
      → ('I^Ω →w 'O^Ω)"
```

Feedback is also causal, as long as every feedback-circle has at least one strongly causal component. Figure 5.4 displays such a composition between weakly causal and strongly causal component.

Notice that no output from g directly goes to the input of g. The following lemma proves the situation:

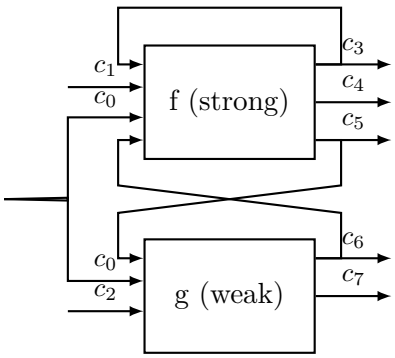

Figure 5.4: Composition of a weakly causal functions with a strongly causal function. The resulting system is weakly causal.

```
lemma spfcomp_feed_weak1:
  fixes f::"'fIn::tchan^Ω → 'fOut::tchan^Ω"
    and g::"'gIn::tchan^Ω → 'gOut::tchan^Ω"
  assumes (* No self-loops for g *)
    "chDom TYPE('gOut) ∩ chDom TYPE('gIn) = {}"
    and "strong f" and "weak g"
  shows "weak (f ⊗ g)"
```

5.1.5 Key Definitions for Deterministic Components

Table 5.1 lists important function over SPFs and TSPFs. Many functions over TSB are also TSPFs. An example is the time-converter to event bundles `convEvent`. It is weakly causal and thus can be used as TSPF. Hence the Table 4.1 over TSB also contains some TSPFs especially useful for constructing new TSPFs.

Restrict Channels These functions restrict the channels. Channels not in the set are excluded from the input/output and contain only the empty stream.

```
definition spfInputRestrict:: "'I set ⇒ ('I^Ω → 'O^Ω) → ('I^Ω → 'O^Ω)"

definition spfOutputRestrict::"'O set ⇒ ('I^Ω → 'O^Ω) → ('I^Ω → 'O^Ω)"
```

The restricting functions allow reusing components, e.g., from a component library, with different interfaces.

Drop Messages Drop the first n time slices of the input/output.

```
definition tspfInputDrop:: "nat ⇒ ('I^Ω → 'O^Ω) → ('I^Ω → 'O^Ω)"

definition tspfOutputDrop::"nat ⇒ ('I^Ω → 'O^Ω) → ('I^Ω → 'O^Ω)"
```

These dropping functions allow specifying a components execution state, e.g., a water tank which is already half full.

Take Messages Take the first n time slices of the input/output.

```
definition tspfInputTake:: "nat ⇒ ('I^Ω → 'O^Ω) → ('I^Ω → 'O^Ω)"

definition tspfOutputTake::"nat ⇒ ('I^Ω → 'O^Ω) → ('I^Ω → 'O^Ω)"
```

These taking functions allow specifying components that only react to a finite input, or produce only a finite output.

Concatenate Messages Add a prefix to the input/output.

```
definition spfInputConc:: "'I^Ω ⇒ ('I^Ω → 'O^Ω) ⇒ 'I^Ω → 'O^Ω"

definition spfOutputConc::"'O^Ω ⇒ ('I^Ω → 'O^Ω) ⇒ 'I^Ω → 'O^Ω"
```

These inserting functions allow customizing components to start with in a specific execution state or to produce an initial output.

Delay Messages Delay the input/output by exactly n time slices.

```
definition tspfDelayInput:: "nat ⇒ ('I^Ω → 'O^Ω) ⇒ 'I^Ω → 'O^Ω"

definition tspfDelayOutput::"nat ⇒ ('I^Ω → 'O^Ω) ⇒ 'I^Ω → 'O^Ω"
```

Delaying functions allow configuration of a component's timing, e.g., a component that only starts working after 10 time-slices.

Underspecified Delay Set of TSPFs with the same untimed behavior, but with variable delays.

```
definition tspfDelayAtMost:: "nat ⇒ ('I^Ω → 'O^Ω) ⇒ ('I^Ω → 'O^Ω) set"

definition tspfDelayAtLeast::"nat ⇒ ('I^Ω → 'O^Ω) ⇒ ('I^Ω → 'O^Ω) set"

definition tspfDelayMinMax:: "nat × nat ⇒ ('I^Ω → 'O^Ω)
                                        ⇒ ('I^Ω → 'O^Ω) set"
```

Underspecified delaying functions allow checking properties even if the concrete timing of components is unknown, e.g., if an alarm is risen after at most 3 time-slices.

Table 5.1: Definitions on TSPF.
Type-Abbreviations are explained in Table 3.1

	Name	Signature	Description	See
	spfDom	$I \rightarrow O \rightarrow \mathbb{P}(channel)$	input channels	[BKR+20]
	spfRan	$I \rightarrow O \rightarrow \mathbb{P}(channel)$	output channels	[BKR+20]
	spfType	$I \rightarrow O \rightarrow \mathbb{P}(channel) \times \mathbb{P}(channel)$	return the type	[BKR+20]
	weak	$I \rightarrow^{\mathtt{t}} O \Rightarrow \mathbb{B}$	ensure weak causality	74
	strong	$I \rightarrow^{\mathtt{t}} O \Rightarrow \mathbb{B}$	ensure strong causality	74
	Abs_tspfw	$I \rightarrow^{\mathtt{t}} O \Rightarrow I \rightarrow^{\mathtt{t}}_{\mathtt{w}} O$	convert to TSPF_w	77
	Rep_tspfw	$I \rightarrow^{\mathtt{t}}_{\mathtt{w}} O \Rightarrow I \rightarrow^{\mathtt{t}} O$	convert to SPF	77
	Abs_tspfs	$I \rightarrow^{\mathtt{t}}_{\mathtt{w}} O \Rightarrow I \rightarrow^{\mathtt{t}}_{\mathtt{s}} O$	convert to TSPF_s	77
	Rep_tspfs	$I \rightarrow^{\mathtt{t}}_{\mathtt{s}} O \Rightarrow I \rightarrow^{\mathtt{t}}_{\mathtt{w}} O$	convert to TSPF_w	77
	Abs_tspf_fun	$I \rightarrow^{\mathtt{t}} O \Rightarrow I \rightarrow^{\mathtt{t}}_{\mathtt{s}} O$	convert to TSPF_s	77
	Rep_tspf_fun	$I \rightarrow^{\mathtt{t}}_{\mathtt{s}} O \rightarrow I \rightarrow^{\mathtt{t}} O$	convert to SPF	77
$\otimes$	**spfComp**	$I \rightarrow O \rightarrow J \rightarrow P \rightarrow ((I \cup J) - (O \cup P)) \rightarrow (O \cup P)$	general composition	78
$\otimes_s$	**spfsComp**	$I \rightarrow^{\mathtt{t}}_{\mathtt{s}} O \rightarrow J \rightarrow^{\mathtt{t}}_{\mathtt{s}} P \rightarrow ((I \cup J) - (O \cup P)) \rightarrow^{\mathtt{t}}_{\mathtt{s}} (O \cup P)$	causal general composition	78
$\parallel$	**spfCompPar**	$I \rightarrow O \rightarrow J \rightarrow P \rightarrow (I \cup J) \rightarrow (O \cup P)$	parallel composition	[BKR+20]
$\parallel_w$	**spfwCompPar**	$I \rightarrow^{\mathtt{t}}_{\mathtt{w}} O \rightarrow J \rightarrow^{\mathtt{t}}_{\mathtt{w}} P \rightarrow (I \cup J) \rightarrow^{\mathtt{t}}_{\mathtt{w}} (O \cup P)$	causal parallel composition	80
$\circ$	**spfCompSeq**	$I \rightarrow L \rightarrow L \rightarrow O \rightarrow I \rightarrow O$	sequential composition	[BKR+20]
$\circ_w$	**spfwCompSeq**	$I \rightarrow^{\mathtt{t}}_{\mathtt{w}} L \rightarrow L \rightarrow^{\mathtt{t}}_{\mathtt{w}} O \rightarrow I \rightarrow^{\mathtt{t}}_{\mathtt{w}} O$	causal seq. composition	80
μ	**spfFeedback**	$I \rightarrow O \rightarrow (I - O) \rightarrow O$	feedback composition	[BKR+20]

Table 5.1 Continued: Definitions on TSPF.
Type-Abbreviations are explained in Table 3.1

Name	Signature		Description	See
spfInputRestrict	$\mathbb{P}(I)$	$\Rightarrow\ I \twoheadrightarrow O \ \rightarrow\ I \twoheadrightarrow O$	restrict input channel	81
spfOutputRestrict	$\mathbb{P}(O)$	$\Rightarrow\ I \twoheadrightarrow O \ \rightarrow\ I \twoheadrightarrow O$	restrict output channel	81
tspfInputDrop	$\mathbb{N}$	$\Rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O \ \rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O$	drop first n time slices	81
tspfOutputDrop	$\mathbb{N}$	$\Rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O \ \rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O$	drop first n time slices	81
tspfInputTake	$\mathbb{N}$	$\Rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O \ \rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O$	take first n time slices	82
tspfOutputTake	$\mathbb{N}$	$\Rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O \ \rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O$	take first n time slices	82
spfInputConc	I^{Ω}	$\Rightarrow\ I \twoheadrightarrow O \ \rightarrow\ I \twoheadrightarrow O$	prefix with bundle	82
spfOutputConc	O^{Ω}	$\Rightarrow\ I \twoheadrightarrow O \ \rightarrow\ I \twoheadrightarrow O$	prefix with bundle	82
tspfDelayInput	$\mathbb{N}$	$\Rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O \ \rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O$	delay by n time slices	82
tspfDelayOutput	$\mathbb{N}$	$\Rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O \ \rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O$	delay by n time slices	82
tspfDelayAtLeast	$\mathbb{N}$	$\Rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O \ \rightarrow\ I \rhd^{t} O$	delay by at least n	82
tspfDelayAtMost	$\mathbb{N}$	$\Rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O \ \rightarrow\ I \rhd^{t} O$	delay by at most n	82
tspfDelayMinMax	$\mathbb{N} \times \mathbb{N}$	$\Rightarrow\ I \twoheadrightarrow^{\mathtt{t}} O \ \rightarrow\ I \rhd^{t} O$	delay by at least *min* and at most *max*	82

5.1.6 Methodology

This chapter discusses different approaches to define the behavior of deterministic components.

Lifting of Non-Continuous Functions

A generic way to create an SPF is by using non-continuous functions. Using the `Abs_cfun` and `Abs_tspfw` operators, these functions can be lifted to a TSPF definition.

```
definition "liftedTSPF = Abs_tspfw (Λ sb.
if(#√sb < 100) then tsbDelay 100·⊥
               else tsbDelay 100·⊥ ^Ω tsbDrop 100·sb)"
```

The example uses the `if-then-else` function, which is non-continuous by default. In order to use the function, continuity and weak causality have to be shown. Proving causality is straightforward due to the existing proofs over the length of `tsbDelay` and `tsbDrop`. Continuity requires more effort and an additional assumption demanding that the bundle may only contain a finite amount of channels.

Discussion: This method should be used as last resort when no other approach can be applied. Proving continuity can be a significant burden and should be avoided if possible.

Composition of Continuous Functions

All functions in Section 5.1.5 already come with continuity proofs. Using these functions instead of manually proving continuity allows for simpler definitions. Only a simple causality proof is required.

```
definition "liftedTSPF2 = Abs_tspfw (sbConc (tsbDelay 100)
                                     oo tsbDrop 100)"
```

Discussion: This is one of the easiest and best ways to define behavior. Due to large number of continuous functions over TSB (see Section 4.3) and SPF (see Section 5.1.5) it is often applicable.

Recursive Definition

Some of the defined functions internally use recursion. For example, `eTake (Suc n)` is defined using `eTake n`. When the developer has a recursive specification, the definition can be realized using the least-fixpoint operator. An example are deterministic automata.

```
definition "liftedTSPF3 = Abs_tspfw (fix·(Λ tspf sb.
                          tsbDelay 1·⊥ ^Ω tspf·(tsbRt·sb)))"
```

Discussion: This method should be used if the specification requires recursion and existing functions over TSB and TSPF are not sufficient. The complexity of the continuity proof depends on the recursive-specification. This is goes back to defining continuous functions as described in the previous two paragraphs. But the causality proof is more difficult, especially when the recursion is not over time slices.

SOME & THE

Compared to the previous approaches, this one is less closely related to normal functional programming definitions. We only write down the requirements of the component. Then we say: "Give me some behavior, which fulfills the requirements". Instead of continuity and causality, this method requires existence proofs.

```
definition "liftedTSPF4 = Abs_tspfw (SOME tspf. weak tspf
      ∧ (∃sb. tspf·sb = sb ⊥ ^Ω sb)
      ∧ (∀sb. tspf·sb ≠ sb))
```

Discussion: Proving existence of a function is generally *significantly* more complicated than proving causality and continuity. Often, existence is proven by constructing a component using one of the approaches above. Hence this approach is only useful in very specific situations where proving existence is easy.

Comparison of Methods

Table 5.2 gives a direct comparison of the presented methods.

Name	Prove Continuity	Prove Weak	Prove Existence	Difficulty of Definition	Expressiveness
Non-Continuous	hard	medium	-	easy	restricted
Comp. Continuous	-	medium	-	easy	restricted
Recursive Definition	easy	medium	-	medium	powerful
SOME&THE	-	-	*very* hard	easy	powerful

Table 5.2: Comparison of different methods to create deterministic components

The following questions can help when defining a new deterministic component.

1. Is there an existing function? See Section 4.3 and Section 5.1.5
2. Is it possible to continually combine existing function, e.g. using sequential composition?

3. Does the component require recursion, which cannot be handled using existing functions?
4. Can I lift a non-continuous functions, e.g., `if-then-else`?
5. Is it possible to prove existence? Then use SOME or THE.

5.2 Underspecified Component Specifications

Early development-phases contain underspecification, for example as a result of incomplete or developing requirements. Underspecification can be modeled with the nondeterministic or underspecified components introduced in this section.

5.2.1 Datatype for Underspecified Component Specifications

We define a timed stream processing specification (TSPS) as a set of TSPFs with fixed input and output channels [Rum96].

Definition 5.2 (timed stream processing specification (TSPS))**.** Let C be the set of all possible channels and $I, O \subseteq C$. We define the weakly causal TSPS type with input channels I and output channels O as shown below:

$$TSPS_{w_{I,O}} := \mathbb{P}(I^{\underline{\Omega}} \rightarrow_w O^{\underline{\Omega}})$$

And strongly causal as:

$$TSPS_{s_{I,O}} := \mathbb{P}(I^{\underline{\Omega}} \rightarrow_s O^{\underline{\Omega}})$$

In Isabelle, the predefined type `set` is used. For example, a strongly causal TSPS between input bundle I^Ω and output bundle O^Ω has the signature $(I^\Omega \rightarrow_s O^\Omega)$ *set*.

SPS variables always start with an uppercase letter, commonly F, G, S or H. The lowercase variable denotes an SPF which is contained in the SPS. e.g, $f \in F$. Both TSPS datatypes contain the empty set {}. This is required to fulfill the lattice-theory assumptions and brings useful properties, such as a complete minus function and a `lfp` operator. However, we often want to ensure that an implementation exists for a component. A component is called *consistent* if and only if it is not empty. Contradictory requirements lead to inconsistent components.

Table 5.3: Definitions on TSPS.
Type-Abbreviations are explained in Table 3.1

	Name	Signature					Description	See
{}	**empty**	$I \rhd O$					empty component	[BKR+20]
	UNIV	$I \rhd O$					greatest component	[BKR+20]
$\in$	**member**	$I \rightarrow\!\!\blacktriangleright O$	$\Rightarrow$	$I \rhd O$	$\Rightarrow$	$\mathbb{B}$	check if SPF is member	[BKR+20]
$\cup$	**union**	$I \rhd O$	$\Rightarrow$	$I \rhd O$	$\Rightarrow$	$I \rhd O$	union over SPS	[BKR+20]
$\cap$	**inter**	$I \rhd O$	$\Rightarrow$	$I \rhd O$	$\Rightarrow$	$I \rhd O$	intersection over SPS	[BKR+20]
	spsDom	$I \rhd O$	$\Rightarrow$	$\mathbb{P}(I)$			input channels of SPS	[BKR+20]
	spsRan	$I \rhd O$	$\Rightarrow$	$\mathbb{P}(O)$			output channels of SPS	[BKR+20]
	spsIO	$I \rhd O$	$\Rightarrow$	$\mathbb{P}(I^\Omega \times O^\Omega)$			SPS to relation	[BKR+20]
	spsIOtoSet	$\mathbb{P}(I^\Omega \times O^\Omega)$	$\Rightarrow$	$I \rhd O$			relation to SPS	[BKR+20]
	spsComplete	$I \rhd O$	$\Rightarrow$	$I \rhd O$			return complete SPS	[BKR+20]
	spsIsComplete	$I \rhd O$	$\Rightarrow$	$\mathbb{B}$			SPS is complete	[BKR+20]
	spsIsConsistent	$I \rhd O$	$\Rightarrow$	$\mathbb{B}$			SPS is not empty	[BKR+20]
$\bigotimes$	**spsComp**	$I \rhd O$	$\Rightarrow$	$J \rhd P$	$\Rightarrow$	$((I \cup J) - (O \cup P)) \rhd (O \cup P)$	general composition	89
$\bigotimes_s$	**spssComp**	$I \rhd^t_s O$	$\Rightarrow$	$J \rhd^t_s P$	$\Rightarrow$	$((I \cup J) - (O \cup P)) \rhd^t_s (O \cup P)$	causal general composition	89
$\parallel$	**spsCompPar**	$I \rhd O$	$\Rightarrow$	$J \rhd P$	$\Rightarrow$	$(I \cup J) \rhd (O \cup P)$	parallel composition	[BKR+20]
$\gg$	**spsCompSeq**	$I \rhd L$	$\Rightarrow$	$L \rhd O$	$\Rightarrow$	$I \rhd O$	sequential composition	[BKR+20]
μ	**spsCompFeed**	$I \rhd O$	$\Rightarrow$	$(I - O) \rhd O$			feedback composition	[BKR+20]
	spsMakeWeak	$I \rightarrow\!\!\blacktriangleright O$	$\rightarrow$	$I \rhd^t_w O$			TSPS_w with same behavior	90
	spsMakeStrong	$I \rightarrow\!\!\blacktriangleright O$	$\rightarrow$	$I \rhd^t_s O$			TSPS_s with same behavior	90

5.2.2 Composition of Component Specifications

Non-determinisitic components are composed via their determinisitc behaviors. Consider two non-deterministic components F and G. Both contain exactly two behaviors. Then the TSPS-composition is the pairwise application of SPF-composition:

$$F = \{f_1, f_2\}$$
$$G = \{g_1, g_2\}$$
$$F \bigotimes G = \{f_1 \otimes g_1, f_1 \otimes g_2, f_2 \otimes g_1, f_2 \otimes g_2\}$$

In Isabelle, composition is abbreviated using $\bigotimes$. The strongly causal composition is denoted as $\bigotimes_s$. The composition is defined as follows:

```
definition spsComp::
  "('I1^Ω → 'O1^Ω) set ⇒ ('I2^Ω → 'O2^Ω) set
     ⇒ ((('I1 ∪ 'I2) - 'O1 ∪ 'O2)^Ω → ('O1 ∪ 'O2)^Ω) set" where
"spsComp F G = {f ⊗ g | f g. f∈F ∧ g∈G }"
```

The composition of two strongly-causal components is a strongly-causal component:

```
definition spssComp::
"('I1^Ω →s 'O1^Ω) set ⇒ ('I2^Ω →s 'O2^Ω) set
    ⇒ ((('I1 ∪ 'I2) - 'O1 ∪ 'O2)^Ω →s ('O1 ∪ 'O2)^Ω) set" where
"spsComp_strong F G = {f ⊗s g | f g. f∈F ∧ g∈G }"
```

Composing two consistent components leads to a consistent component. This is important because, as a result, it suffices to check each atomic component for consistency. The consistency of the composition follows automatically:

```
theorem spscomp_consistent:
  assumes "F ≠ {}"
      and "G ≠ {}"
    shows "(F ⊗ G) ≠ {}"
```

For sequential composition of components the infix-operator $\gg$ is created. The output of the left component is fed into the right component.

```
lemma "sps1 ≫ sps2
     = {spf2 oo spf1 | spf1 spf2. spf1 ∈ sps1 ∧ spf2 ∈ sps2}"
```

5.2.3 Refinement of Component Specifications

In the idealized development process, the first version of a component specification is an abstract specification with underspecification. In multiple refinement steps this abstract specification is modified to a concrete specification. Focus [BS01b] defines refinement

as the subset-relationship ($\subseteq$) between components. When a property P holds over each deterministic-behavior in a component F, then it also holds over each deterministic behavior in the refinement *F_ref*:

```
theorem
  assumes "∀f∈F. P f"
      and "F_ref ⊆ F"
  shows "∀f_ref∈F_ref. P f_ref"
```

Refinement of a component in a decomposed structure automatically leads to refinement of the composition [BR07].

```
theorem spscomp_refinement:
  assumes "F_ref ⊆ F"
      and "G_ref ⊆ G"
    shows "(F_ref ⊗ G_ref) ⊆ (F ⊗ G)"
```

5.2.4 Key Definitions for Underspecified Components

Table 5.3 lists important function over SPS and TSPS.

Convert to Causal Specification These functions take an untimed SPF and return a weakly/strongly TSPS with the same untimed behavior.

```
definition spsMakeWeak:: "('a untimed^Ω → 'b untimed^Ω)
                          ⇒ ('a^Ω →_w 'b^Ω) set"

definition spsMakeStrong:: "('a untimed^Ω → 'b untimed^Ω)
                          ⇒ ('a^Ω →_s 'b^Ω) set"
```

5.2.5 Methodology

This section discusses different approaches to defining the behavior of non-deterministic components.

Listing Deterministic Functions

The simplest variant is to explicitly build the set of SPFs by listing the containing deterministic functions. Different methods to create deterministic functions are presented in Section 5.1.6. An example is the following TSPS, which is either the identity or delays the input by one time slice:

```
definition "tsps = {ID, tspfDelayOutput 1 ID}"
```

Discussion: Explicitly listing the behavior is the easiest way to define a non-deterministic component. Determining whether a deterministic behavior is contained in the SPS is straightforward. Additionally, one can perform case distinctions to prove properties over the SPS. The consistency follows directly. But since the underspecification is limited, this method is not generally usable in early development stages where components are often heavily underspecified, i.e., the set of functions is infinitely large.

Set Comprehension

This method allows components with infinite different behaviors. For example, the following `tsps_delay` contains functions which delay more than 100 time slices.

```
definition "tsps_delay = {tspfDelayOutput n ID | n::nat. n>100}"
```

Here, n is a natural number, hence `tsps_delay` only contains countably many behaviors. When using a larger type, the behavior can have uncountably infinite elements. The following listings uses $\mathbb{N}^\infty$:

```
definition "tsps_ora = {tspfAdd·oracle
                          | oracle::nat stream. #oracle = ∞}"
```

Because the variable can reduce non-determinism in the component, it is also referred to as an *oracle*. If the oracle is a stream, it can be viewed as an additional input channel of the component.

Discussion: Note that the oracle is fully known before any input is passed to the component. For many behaviors, it is desirable that the non-deterministic choice depends on previous inputs. Showing consistency is relatively easy, one needs to find a single valid oracle.

Relational Specification

This variant uses a predicate which restricts the allowed behaviors. This predicate can be viewed as a relational specification.

```
definition rel2sps::"('I ⇒ 'O ⇒ bool) ⇒ ('I → 'O) set" where
"rel2sps P = {spf | spf. ∀input. P input (spf·input)}"
```

On first glance, this looks similar to the "Set Comprehension" method above. But here the variable is a function, and not a natural number or a stream. This allows significantly different ways to define behavior. Hence, it is a separate method. The following component contains all weakly causal functions, which arbitrarily delay the input:

```
definition "tsps = rel2sps (λi out. #√i ≤ #√out
                              ∧ convUntimed·i ⊑ convUntimed·out)"
```

While the user-supplied specification is relational, the resulting SPS is always a set of functions. If the user defines a non-continuous predicate, the empty SPS is returned. An example is:

$$P_{invalid}\ input\ output := \text{if } (\#_{\surd} input < \infty) \text{ then } output = \epsilon \text{ else } output = input$$

Discussion: Recursive definitions are not possible. Consistency is harder to show than in the previous methods. To prove consistency, one can manually define a deterministic behavior and show that the behavior fulfills the predicate.

Recursive Specification

The previous two methods where unable to handle recursive definitions of SPS. An example of a recursive definition is:

$$TSPS_{rec} = \{tspfInputDrop\ 1\ (tspfOutputDrop\ 1\ tspf)) \mid tspf \in TSPS_{rec}\}$$

The recursive equation has multiple fixpoints. For example, the lfp is the empty set. To fully capture underspecification, greatest fixpoint (gfp) is used. This ensures that the most general SPS is returned. To prove the existence of a fixpoint, monotonicity has to be proven.

```
definition "tsps_gfp = gfp(λTSPS.
      (λtspf. tspfInputDrop 1·(tspfOutputDrop 1·tspf)) `TSPS)"
```

Discussion: If recursive specifications are required, then this approach is appropriate. However, these specifications can be hard to fully understand, because the gfp may contain unexpected behavior. Lemmata of the form $X \subseteq gfp\ F$ are generally easier to show than $gfp\ F \subseteq Y$. Proving consistency of the fixpoint works similar to the relational specification described above.

Set Operators

Set-operators can also be used to specify behavior. For example $F \cap G$ is the largest component that both fulfills the requirements of F and G. With the minus-Operator $(-)$ sets can be inverted and behavior forbidden. When *FORBIDDEN* is the SPS with undesirable behaviors, $UNIV - FORBIDDEN$ is a new SPS where the undesirable behavior is forbidden.

Discussion: This method is suitable for combining and modifying existing SPS. However, it quickly reaches its limits for new requirements. It is a method for combining existing components, and not creating completely new components.

Comparison of Methods

Table 5.4 gives a direct comparison of the presented methods.

Name	Prove Consistency	Prove Mono	Difficulty of Definition	Expressive-ness	Under-specification
Listing Functions	easy	-	easy	limited	limited
Set Comprehension	medium	-	medium	medium	medium
Relational Spec	hard	-	easy	limited	powerful
Recursive Spec	hard	medium	hard	powerful	medium
Set Operator	medium	-	easy	limited	medium

Table 5.4: Comparison of different methods to create non-deterministic components

The following questions guide you to the right specification method:

1. Is the component best described by a finite number of easily definable SPF? Explicitly define the individual SPFs and use the method *Listing Deterministic Functions*.
2. Are there existing SPSs which can be re-used and combined? Use set-operators like $(\cup)$, $(\cap)$, or $(-)$ as shown in method *Set Operators*.
3. Can you use an simple oracle to define the underspecification? Then use the method *Set Comprehension*.
4. Is it possible to describe the behavior as relation between inputs and outputs? Use the method *Relational Specification*.
5. Do you need recursion to define the specification? Employ the *Recursive Specification* method.

Chapter 6

Timed Case Study in Isabelle

We demonstrate the usefulness of the definitions and proof strategies from the previous chapters in a comprehensive case study. For this purpose, we chose a road speed monitoring system. Speeding has been one of the leading causes of traffic accidents ever since modern vehicles were brought to practical use. Speed cameras have become a popular tool for law enforcement to detect and punish speeding. These cameras are typically mounted over or beside the roads and captures relevant data such as driving speed and license plates. Posting cameras and processing large amounts of traffic data poses a great challenge however. One factor are law enforcement human resources, i.e., officers bound up in photo collection, reviewing potential violations, and following up with traffic citations. Additional factors include storage limitations, the risk of false positives, data-privacy, and cyber-security. An automated yet safe and secure speed monitoring system thus is beneficial in ensuring future road safety.

The system is designed to capture speed measurements along with the corresponding license plate information from traffic data. It will serve as a platform for data retrieval, enabling law enforcement to issue citations efficiently. As such, it shall provide easy access to data about speeding incidents. Internally, the system shall use a standard speed camera component. Furthermore, data collected by the camera shall be preserved for potential subsequent processing. To this end, a remote database shall be connected to the camera through wireless data transmission. As traffic data is vast and storage is limited, not all traffic data but only speeding violations shall be stored in the database. Law enforcement agents require the violations to be accessible in a straight-forward way. The database shall therefore be queried through a user-oriented interface without knowing the internal database structure.

Due to the distributed nature of the case study requirements, the system is specified as a Focus composition of re-usable components. The overall system takes take two different inputs. One for the traffic data to be captured by the speed camera. This traffic data includes both speed and license plate data. The second input is for inquiries into past violations that are stored in the data base. These inquiries are made in the form of a license plate. The overall system has a single output for results of violation inquiries. The form of the results are the retrieved license plate data coupled with potential speeding violations. Internally, the system is composed of a speed camera, a database, and an

external Application Programming Interfaces (API). The camera observes traffic data and sends violations to the database. The database stores violations and provides a database-specific interface for requesting data. The API takes data inquiry requests from law enforcement, runs the database-specific requests against the internal database, and returns the response in the required form. Figure 6.1 gives an overview of the resulting distributed speed monitoring system, its data channels and internal components.

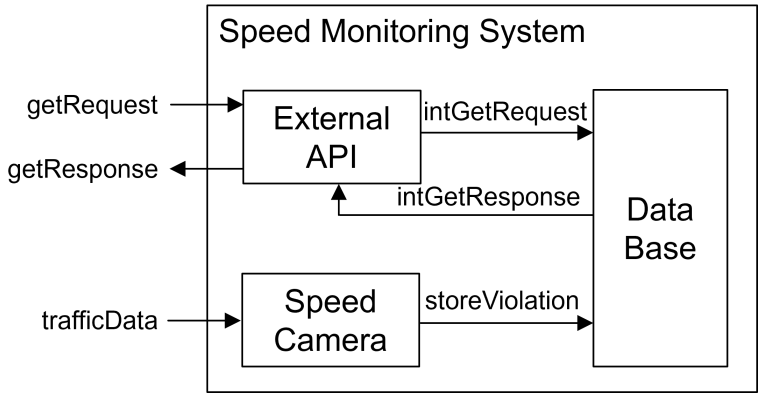

Figure 6.1: Overview of the Speed Monitoring System: The system observes traffic (`trafficData`) and records `violations` through a speed camera component. Law enforcement is able to `getRequest` past violations via an external API. As a `getResponse` to requests, potential violations are retrieved from the database through internal, database-specific requests (`intGetRequest`) and responses (`intGetResponse`).

In this case study, we specify and verify the speed monitoring system in Isabelle with the concept of Focus. The formal specification and verification of this system are critical. Verification ensures correct operation (safety) of the system, i.e., that the data it captures and stores are accurate and reliable. This is necessary as incorrect data can result in incorrect citations and unjust penalties. Furthermore, the system shall be secure against attacks, such as malicious requests. Potentially recorded data of sensitive vehicles shall not be available for inquiry by untrusted parties as it could lead to the leak of confidential information. The system is expected to respond with an empty (null) answer within a fixed time frame. This is necessary for non-malicious users to be able to distinguish between forbidden requests and responses lost in transmission.

6.1 Channel and Message Datatypes

The speed monitoring system is composed of three components that are interconnected to each other and the outside world by six channels. To better understand the channels'

purpose, here is an exhaustive list with additional information about the messages they carry:

- `trafficData`: Tuples of license plate data and speed per passing vehicle. Observed by the speed camera;
- `storeViolation`: License plate data, speed, and photo of speeding vehicles as determined by the speed camera. Transmitted to the database component;
- `getRequest`: Law enforcement inquiries into license plates sent to the API;
- `intGetRequest`: Inquiries forwarded by the API internally to the database;
- `intGetResponse`: Potential records of speed violations retrieved from the database, as tuple of license plate, speed, and the photo. Sent to the API;
- `getResponse`: Potential records of speed violations returned to law enforcement. Photos are excluded for privacy reasons.

After conceptually laying out the channels and message types, they need to be encoded in Isabelle. The speed monitoring system requires the following three raw data types:

```
datatype licensePlate = LicencePlate string
datatype speed        = Speed rat
datatype photo        = Photo "(char list) list"
```

License plates are encoded as strings, speeds of a car as rational numbers, and a photograph is encoded as an two dimensional array containing color values for the pixels.

The message datatype, `msg`, is the underlying type for all channels communication messages. It contains no timing information as this kind of information is encoded in the stream type. The `msg` type is, in the context of the speed monitoring system, defined as follows:

```
datatype msg =
  MEASURE   "licensePlate × speed"                    |
  STORE     "licensePlate × (speed × photo)"          |
  GET       "licensePlate"                            |
  INTERNAL  "licensePlate × ((speed × photo) option)" |
  RESULT    "licensePlate × (speed option)"
```

The interfaces of the components and the overall speed monitoring system can now be formally specified in terms of Focus bundles:

- external API: $\{\texttt{getRequest, intGetResponse}\}^{\Omega} \rightarrow_w \{\texttt{intGetRequest, getResponse}\}^{\Omega}$;
- speed camera: $\{\texttt{trafficData}\}^{\Omega} \rightarrow_w \{\texttt{storeViolation}\}^{\Omega}$;

- database: {intGetRequest, storeViolation}$^{\Omega}$ $\rightarrow_w$ {intGetResponse}$^{\Omega}$;
- speed monitoring system: {getRequest, trafficData}$^{\Omega}$ $\rightarrow_w$ {getResponse}$^{\Omega}$.

The channels are typed, i.e., allow only specific messages to be transmitted:

- cType getRequest = cType intGetRequest $\subseteq$ range GET
- cType intGetResponse $\subseteq$ range INTERNAL
- cType getResponse $\subseteq$ range RESULT
- cType trafficData $\subseteq$ range MEASURE
- cType storeViolation $\subseteq$ range STORE

For each of theses sets we define a new type.

```
typedef InAPI = "{getRequest, intGetResponse}"
typedef OutAPI = "{getResponse, intGetRequest}"
typedef InCAM = "{trafficData}"
typedef OutCAM = "{storeViolation}"
typedef InDB = "{storeViolation, intGetRequest}"
typedef OutDB = "{intGetResponse}"
typedef InSys = "{getRequest, trafficData}"
typedef OutSys = "{getResponse}"
```

In this case study, all channels histories are event streams. Thus, we require and instantiate the newly created types to be in the echan class.

```
instantiation InAPI :: "{echan}"
...
```

The stream bundles do not directly map channels to streams of raw data types like licensePlate, speed, or photo. Instead, they map combined streams of message type msg. We leverage the getter function to explicitly convert msg combined-stream to the concrete types, e.g., an licensePlate event stream. This getter function (and its inverse) are lifted to the SB level and able to get the concrete streams of a SB and the inverse, i.e., defining a SB by providing the concrete streams instead of *msg combinedstreams*. The verification of the system is conducted on the stream level. The sbsetter and sbgetter functions for each case-study channel domain type abstract from irrelevant bundling.

6.2 System Specifications

The speed monitoring system shall be safe (operate according to the wanted behavior) and secure (does not allow for unwanted behavior). To formally verify these properties, we begin by encoding them formally. We achieve this via predicates on input/output streams and on parameters such as allowed speeds and public license plates. We then also specify the behavior of the subsystems, API, speed camera, and database. Finally, we'll demonstrate a successful refinement proof between the safety & security properties and the composed system of distributed subsystems. This guarantees safety and security for the composition of subsystems.

Safety and Security Requirements

Based on the informal requirements in the beginning of this chapter, we develop the following safety and security requirements. For all gathered traffic data, all inquiries, and all responses of the system, it should hold that:

- `safety_1`: The response matches the traffic data, i.e., the speed camera captured the vehicle in question at the indicated speed;
- `safety_2`: The vehicle in question was speeding;
- `safety_3`: Each response corresponds to a request, nothing is answered unwarrantedly;
- `safety_4`: Unauthorized requests are responded to by a null reponse within two time intervals;
- `security_1`: Only requests for public plates are answered, i.e., responses are filtered according to a whitelist.

The first safety requirement is derived directly from the requirement for reliability: the responses given to law enforcement should match actually captured traffic data. The system must not create false readings. The formalization below captures `safety_1` via a predicate over the traffic data and responses given by the system. It requires that all responses of plates and speeds at any point `n` match a traffic data observation at some point `m`. Precisely, it states that for any ($\forall$) plate, speed, and index `n` such that the following sub-predicate must hold. The index `n` being below ($<$) the number of events (`#`) recorded in the stream of responses and ($\wedge$) the `n`-th event (`snth n`) of the response stream being equal to ($=$) a message (`Event` constructor) recording the aforementioned plate and non-empty (`Some` constructor of the `option` datatype) speed implies ($\longrightarrow$) the f following sub-sub-predicate. There exists ($\exists$) an index `m` such that `m` is below the

number of traffic data events and the m-th traffic data element is a message carrying aforementioned plate and speed:

```
definition safety_1 :: "(licensePlate × speed) event stream
          ⇒ (licensePlate × speed option) event stream
          ⇒ bool" where
"safety_1 trafficData response ≡ ∀plate speed n.
   n < #response ∧ snth n response = Event (plate, Some speed)
     ⟶ (∃m. m < #trafficData
        ∧ snth m trafficData = Event (plate, speed))"
```

Considering the limited database storage, the system shall only store and thus return data from speeding violations. The formalization of `safety_2` is thus a predicate over the legal speed limit and responses. It requires all responses of plates and speeds at any point `n` to indicate a violation, i.e., the recorded speed to be above the legal speed limit. Its formalization requires that for any plate, speed, and index `n` the following must hold. The index `n` being below the number of response events and the `n`-th response being a message recording plate and non-empty speed implies the recorded speed to be above ($>$) the legal speed limit:

```
definition safety_2 :: "speed
          ⇒ (licensePlate × speed option) event stream
          ⇒ bool" where
"safety_2 speedLimit response ≡ ∀plate speed n.
   n < #response ∧ snth n response = Event (plate, Some speed)
     ⟶ speed > speedLimit"
```

The system shall only respond to requests, i.e., not unwarrantedly return traffic violations. This requirement is vital for preventing information leaks. The formalization of `safety_3` is thus a predicate over the requests and responses. It requires all plates of all responses at any point `n` to match the plate of a request. Its formalization requires that for any plate, optional speed, and index `n` the following must hold. The index `n` being below the number of response events and the `n`-th response being a message recording plate and non-empty speed implies the existence of an index `m` such that `m` is below the number of request events and the `m`-th request being a message recording the aforementioned plate:

```
definition safety_3 :: "licensePlate event stream
          ⇒ (licensePlate × speed option) event stream
          ⇒ bool" where
"safety_3 request response ≡ ∀n optionalspeed plate.
   n < #response ∧ snth n response = Event (plate, optionalspeed)
     ⟶ (∃m. m < #request ∧ snth m request = Event plate)"
```

As declared before, an empty reply is returned by the system for any non-allowed inquiries in maximal two time-units after their input. For this additional timed property, we again define a predicate in Isabelle over the whitelist of plates, the stream of

requests, and the stream of responses. It requires that for any index n and any plate, the following sub-predicate must hold. The plate being part (∈) of the set of distinct messages (sValues) of the substream at time slice n (esnth n) of the requests and the plate not being part of (∉) the whitelist of public plates implies the existence of a timeout such that the timeout is below 2 and an empty message (tuple with second element None) is part of the set of distinct responses at time slice n plus (+) the timeout:

```
definition safety_4 :: "licensePlate set
          ⇒ licensePlate event stream
          ⇒ (licensePlate × speed option) event stream
          ⇒ bool" where
"safety_4 publicPlates request response ≡
  ∀n plate.
    (plate ∈ sValues·(esnth n·request) ∧ plate ∉ publicPlates)
     ⟶ (∃TimeOut. TimeOut <= 2
        ∧ (plate, None) ∈ sValues·(esnth (n +
            TimeOut)·response))"
```

Only plates from the whitelist, i.e., public plates are allowed to be effectively queried. Other plates may be queried by oblivious agents or malicious third parties, but do not return any speeding violations. The following security predicate states that any plate that was returned with any speed at any point n needs to be in the whitelist, i.e., the publicPlates. It requires that for any index n, any plate, and any speed the following sub-predicate must hold. The index n being below the number of response events and the n-th response event being a message recording aforementioned plate and speed implies that the plate is part of the whitelist of public plates:

```
definition security_1 :: "licensePlate set
          ⇒ (licensePlate × speed) event stream
          ⇒ licensePlate event stream
          ⇒ (licensePlate × speed option) event stream
          ⇒ bool" where
"security_1 publicPlates trafficData request response ≡
  ∀n plate speed.
    n < #response ∧ snth n response = Event (plate, Some speed)
    ⟶ plate ∈ publicPlates"
```

We specify a parameterized SPS to combine the individual safety and security requirements. These parameters are the legal speed limit and a whitelist of public license plates. Then the set of all SPFs with signature $(\mathtt{InSys}^{\Omega} \rightarrow_{w} \mathtt{OutSys}^{\Omega})$ set is constrained by applying the individual requirements, i.e., filtering according to the predicates. Additionally, the SPFs are required to be weakly causal, not only for its realizability. Weak causality is necessary for correct specification of the speed monitoring system: traffic data needs to be captured before violations can be recorded, violations in turn need to be recorded before requests can be made, and finally requests to the database need to

be made before matching responses to law enforcement can be given. The resulting set of functions is abstracted to the type of weakly causal functions, TSPF_w:

```
definition SysReqSPS:: "speed ⇒ licensePlate set
                          ⇒ (InSys^Ω →_w OutSys^Ω) set" where
"SysReqSPS speedLimit publicPlates
  = Abs_tspfw ` { f | f. ∀(input::InSys^Ω).
      let getRequest = fst (bundleInSys.sbgetter·input)
          trafficData = snd (bundleInSys.sbgetter·input)
          getResponse = bundleOutSys.sbgetter·(f·input)
      in safety_1 trafficData getResponse
       ∧ safety_2 speedLimit getResponse
       ∧ safety_3 getRequest getResponse
       ∧ safety_4 publicPlates getRequest getResponse
       ∧ security_1 publicPlates trafficData getRequest
           getResponse
       ∧ weak f }"
```

Note the usage of an all-quantified input. This ensures safety and security under all possible circumstances. As the input is a bundle, we employ the getter function `bundleInSys.sbgetter` to retrieve a tuple of streams. The elements in the tuple correspond to channels in the bundles domain type, `InSys`. The first element, retrieved with `fst`, is thus the stream of requests and the second, retrieved with `snd`, the stream of traffic data. Retrieving the stream of responses requires application of the function `f` to the input and usage of the getter function `bundleOutSys.sbgetter` for output domain `OutSys`. To express the causality, we can either simply define the specification as the set of general continuous functions and then add the logical regulation regarding its causality into the definition or leverage the specific function type of causal and continuous functions, namely `'a` $\rightarrow_w$ `'b`. In this case study, we have defined the stream processing specification of the individual components in the former manner to reduce the complexity of low-level verifications. The specification of the system requirements, as well as the composite of component specifications, is, however, be given as sets of `'a` $\rightarrow_w$ `'b`s as above. The abstraction function `Abs_tspfw` is used to abstract a general continuous function to a weakly causal function. Vice versa, `Rep_tspfw` converts a weakly causal function backwards.

In the following we introduce the individual components that are meant to be composed into a complete functioning system. Then, we need to explicitly verify that the composition of the components do satisfy the system requirements.

Speed Camera Component

The speed camera serves the purpose to deliver the captured data of speeding vehicles. For a properly functioning camera, we ask that only data from speeding vehicles be forwarded to the database. Further, it should hold that violations shall not be fa

transmitted data, i.e., license plate and measured speed, should be previously recorded from traffic data. The component shall thus obey the following safety, i.e., correctness, constraints:

- `camera_safety_1`: Violations shall only record vehicles speeding;
- `camera_safety_2`: Violations shall be consistent with traffic data factual, i.e., may only be issued as a result from previously recorded plates and speeds;

The first safety constraint is formulated as a predicate over the legal speed limit and the stream of recorded violations. It requires that any index `n`, plate, speed, and photo are required to obey the following sub-predicate. The index `n` being below the number of Events recorded on the stream of violations and the `n`-th violation event being equal to a message recording said plate, speed, and photo implies the recorded speed being above the legal speed limit:

```
definition camera_safety_1 ::
    "speed ⇒ (licensePlate × (speed × photo)) event stream
           ⇒ bool" where
"camera_safety_1 speedLimit violation =
  ∀n plate speed photo.
    ( n < #violation
      ∧ snth n violation = Event (plate, (speed, photo)) )
    ⟶ speed > speedLimit"
```

The second safety constraint is formulated as a predicate over the stream of traffic data and the stream of violations. It requires that any index, plate, speed, and photo are required to obey the following predicate. The index being below the number of events recorded on the stream of violations and the `n`-th violation being equal to a message recording said plate, speed, and photo implies the existence of an index m such that the following holds. The index `m` must be below the number of events recorded in the stream of traffic data and the `m`-th element in the stream of traffic data is equal to a message carrying said plate and speed:

```
definition camera_safety_2 ::
    "(licensePlate × speed) event stream
   ⇒ (licensePlate × (speed × photo)) event stream ⇒ bool" where
"camera_safety_2 trafficData violation =
  ∀n plate speed photo.
    n < #violation
      ∧ snth n violation = Event (plate, (speed,photo))
        ⟶ ∃m.
          m < #trafficData
            ∧ snth m trafficData = Event (plate, speed)"
```

We specify the camera as a conjunction of the above safety constraints and additional weak causality. Similar to the specification of the overall system in Section 6.2, the speed camera component is thus defined as follows:

```
definition camSPS :: "speed ⇒ (InCAM^Ω →_w OutCAM^Ω)set" where
"camSPS speedLimit = Abs_tspfw {f | f.  ∀(input::InCAM^Ω).
  let trafficData = bundleInCAM.sbgetter·input
      violation = bundleOutCAM.sbgetter·(f·input)
  in camera_safety_1 speedLimit violation
   ∧ camera_safety_2 trafficData violation
   ∧ weak f }"
```

Database Component

The database receives and stores the data input from the speed camera, and answers the incoming inquiries through the user interface either with the stored data or with an empty reply. We expect the component to respond to inquiries in the same order as they have arrived. This is essential to ensure the punctuality of data output. Further, the component should not fabricate data points. Additionally, the component should immediately return an empty answer to non-allowed inquiries immediately with an empty answer. This requirement plays a vital role of realizing the timed system requirement for handling non-allowed inquiries, `safety_4`. Finally, the database is expected to be deployed in a feedback loop, i.e., it does not simply forward received data but responds to incoming inquiries. Feedback requires the introduction of strong causality to establish well-defined semantics. The security constraints thus are summarized as:

- `db_safety_1`: The order of responses shall be identical to the order of the requests;
- `db_safety_2`: The responses shall match a recorded violation;
- `db_safety_3`: Non-allowed requests shall be responded to immidiately with an empty answer, i.e., the response shall be sent in the immidiate next time slice compared to the request;
- `db_security_1`: Only requests for public plates shall be reponded to with speeding violations, i.e., responses are filtered according to a whitelist.
- `causality`: The component shall be strong causal, i.e., responses to requests shall be sent the earliest in the next time slice.

The first safety constraint is formulated as a predicate over the request stream and the response stream. It requires that at some time in the future and abstracting from all timing constraints, the responses' plates element-wise match the plates from the

requests. Note how there is no limit on the delay the system might produce between request and response, only that eventually the system produces an answer. This is formalized by implying (right hand side of the arrow) this match only if infinitely many time slices ($\#_{\surd}$) have passed on the stream of requests. The match itself is specified by removing of all timing information (`e2ustream`) from the request and response streams and comparing the requests (carrying license plate data) to the element-wise (`smap`) first element (`fst`) of the responses (carrying tuples of license plate data, optional speed, and optional photo):

```
definition db_safety_1 ::
    "(licensePlate × speed × photo) event stream
   ⇒ licensePlate event stream
   ⇒ (licensePlate × ((speed × photo) option)) event stream ⇒
       bool" where
"db_safety_1 request response =
  #√request = ∞
    ⟶ e2ustream·request = smap fst·(e2ustream·response)"
```

The second safety constraint is formulated as a predicate over the violation stream and the response stream. It requires that any index `n`, plate, and result are required to obey the following predicate. The index `n` being below the number of events in the response stream and the `n`-th element of the response being equal to a message carrying aforementioned plate and non-empty result implies the existence of an index `m` such that `m` is below the number of events in the stream of violations and the `m`-th element of the violation stream is a message recording the aforementioned plate and result:

```
definition db_safety_2 ::
      "(licensePlate × speed × photo) event stream
     ⇒ (licensePlate × ((speed × photo) option)) event stream
    ⇒ bool" where
"db_safety_2 violation response =
  ∀n plate result.
    ( n < #response
      ∧ snth n response = Event (plate, Some result) )
    ⟶ ( ∃m. m < #violation
         ∧ snth m violation = Event (plate, result) )"
```

The third safety constraint is formulated as a predicate over whitelist of public plates, the stream of requests, and the stream of responses. It requires that for any time slice `t` and any plate, the following predicate must hold. The plate not being part of the whitelist of public plates and the plate being part of the set of distinct elements of the substream at time slice `t` implies that an empty response, i.e., a tuple where the first element is the aforementioned plate and the speed and photo are missing (`None`) is part of the set of distinct elements of the substream at time `t+1` of the response stream:

```
definition db_safety_3 :: "licensePlate set
```

```
    ⇒ licensePlate event stream
    ⇒ (licensePlate × ((speed × photo) option)) event stream ⇒
        bool" where
"db_safety_3 publicPlates request response =
  ∀t plate. plate ∉ publicPlates ∧ plate ∈ sValues·(esnth
     t·request)
    ⟶ (plate, None) ∈ sValues·(esnth (t+1)·response)))"
```

The security constraint is formulated as a predicate over whitelist of public plates and the stream of responses. It requires that for any index n, any plate, and any result, the following predicate must hold. The index n being below the number of elements in the response stream and the n-th element in the response stream being equal to a message carrying aforementioned plate and non-empty result implies the plate being in the whitelist of public plates:

```
definition db_security_1 :: "licensePlate set
    ⇒ (licensePlate × ((speed × photo) option)) event stream ⇒
        bool" where
"db_security_1 publicPlates response =
∀n plate result.
  n < #response ∧ snth n response = Event (plate, Some result)
    ⟶ plate ∈ publicPlates"
```

The stream processing specification of the database is then defined as the conjunction of the above safety and security constraints, as well as the strong causality requirement:

```
definition dbSPS :: "licensePlate set ⇒ (InDB^Ω →s OutDB^Ω) set"
   where
"dbSPS publicPlates
  = Abs_tspfs ` { f | f. ∀(sb::InDB^Ω).
      let violation = fst (bundleInDB.sbgetter·input)
          getRequest = snd (bundleInDB.sbgetter·input)
          getResponse = bundleOutDB.sbgetter·(f·input)
      in db_safety_1 getRequest getResponse
       ∧ db_safety_2 violation getResponse
       ∧ db_safety_3 publicPlates getRequest getResponse
       ∧ db_security_1 publicPlates getResponse
       ∧ strong f }"
```

External API

The external API abstracts the internal database interface and provides users with means to request speeding information. It forwards inquiries to the database and translates the responses of the database back to the user. If a request could not return a license plate, e.g., if the vehicle did not speed or the plate is not available for inquiry, an empty answer is returned in a maximum of two units of time. As we recall, the database responds to

requests with a delay of one unit of time as it is strong causal. We thus ensure the time constraint by requiring the processing time of empty internal responses to be below two units of time. The data must also be inquired by users to prevent information leak, and be given directly from the database without modifications or being created by others. The security constraints thus are summarized as:

- `api_safety_1`: Any non-empty responses to the user shall match a request made by the user;
- `api_safety_2`: Any non-empty responses to the user shall match an internal response from the database, i.e., plates and speeds shall not be altered in any way;
- `api_safety_3`: Any request made by the user shall be forwarded immediately to the database, i.e, in the same unit of time;
- `api_safety_4`: Any empty response shall be processed immediately, i.e., the response sent to the user within the same unit of time.

The first safety constraint is formulated as a predicate over the requests made by and the response given to the user. It states that any index `n`, plate, and speed are required to obey the following sub-predicate. The index `n` being below the number of events in the response stream and the `n`-element of the response stream being equal to a message carrying aforementioned plate and speed implies the existence of an index `m` such that `m` is below the number of events on the request stream and the `m`-th element on the request stream is a message requesting information about aforementioned plate:

```
definition api_safety_1 :: "licensePlate event stream
    ⇒ (licensePlate × (speed option)) event stream ⇒ bool" where
"api_safety_1 request response =
∀n plate speed.
  n < #response ∧ snth n response = Event (plate, Some speed)
⟶ ∃m. m < #request ∧ snth m request = Event plate"
```

The second safety constraint is formulated as a predicate over the internal responses from the database, as well as the responses returned to the user. It states that any index `n`, plate, and speed are required to obey the following sub-predicate. The index `n` being below the number of events in the response stream and the `n`-element of the response stream being a message carrying aforementioned plate and speed implies the existence of an index `m` and a photo such that `m` is below the number of elements in the internal response stream and the `m`-th element in the internal response stream is a message recording aforementioned plate and speed with accompanying photo:

```
definition api_safety_2 ::
    "(licensePlate × ((speed × photo) option)) event stream
   ⇒ (licensePlate × (speed option)) event stream ⇒ bool" where
```

```
"api_safety_2 intGetResponse response =
∀n plate speed.
  n < #response ∧ snth n response = Event (plate, Some speed)
⟶ ( ∃m.
      m < #intGetResponse
      ∧ snth m intGetResponse = Event (plate, Some (speed,
          photo)) )"
```

The third safety constraint is formulated as a predicate over the requests made by users and the internal requests forwarded to the database. It states that any index n and plate to obey the following sub-predicate. A plate being part of the set of requests received in the n-th time slice implies the plate being part of the set of internal requests forwarded within the same n-th time slice:

```
definition api_safety_3 :: "licensePlate event stream
                    ⇒ licensePlate event stream ⇒ bool" where
"api_safety_3 request intGetRequest =
∀n plate. plate ∈ sValues·(esnth n·request)
          ⟶ plate ∈ sValues·(esnth n·intGetRequest)"

definition api_safety_4 ::
    "(licensePlate × ((speed × photo) option)) event stream
   ⇒ (licensePlate × (speed option)) event stream ⇒ bool" where
"api_safety_4 intGetResponse response =
∀n plate.
  (plate, None) ∈ sValues·(esnth n·intGetResponse)
⟶ ( ∃TimeOut.
      TimeOut <= 1
      ∧ (plate, None) ∈ sValues·(esnth (n + TimeOut)·response) )"
```

The API component is additionally specified to be weak causal, ensuring realizability. As before, the stream processing specification of the external API is then defined as the conjunction of the above safety constraints, as well as the weak causality requirement:

```
definition exApiSPS :: "(InAPI^Ω →_w OutAPI^Ω) set" where
"exApiSPS = Abs_tspfw `{ f | f. ∀(sb::InAPI^Ω).
    let request = fst (bundleInAPI.sbgetter·input)
        intGetResponse = snd (bundleInAPI.sbgetter·input)
        intGetRequest = fst (bundleOutAPI.sbgetter·(f·input))
        getResponse = snd (bundleOutAPI.sbgetter·(f·input))
    in api_safety_1 request getResponse
     ∧ api_safety_2 intGetResponse getResponse
     ∧ api_safety_3 request intGetRequest
     ∧ api_safety_4 intGetResponse getResponse
     ∧ weak f}"
```

6.3 Validity of the Composition

In this section we show the validity of the composition of subsystems wrt. the system requirements. This means proving a refinement theorem encoded via a subset relation between the SPS of the overall system, `SysReqSPS`, and the SPS resulting from composing the three subsystems, speed camera, database, and external API. We thus begin by defining said composition of `exApiSPS`, `camSPS`, and `dbSPS` as follows:

```
definition compSPS :: "speed ⇒ licensePlate set
                          ⇒ (InSys^Ω ⇒_w OutSys^Ω) set" where
"compSPS speedLimit publicPlates = Abs_tspfw ` ( spsConvert·(
  exApiSPS ⊗ (camSPS speedLimit) ⊗ (dbSPS publicPlates)
) )"
```

The function `spsConvert` is used to convert the direct composite of specifications to a SPS of the entire system with the signature `(InSys^Ω →_w OutSys^Ω) set`, in order to simplify the later verification, where we perform the proof of set inclusion between this composite and the `SysReqSPS`. This process also improves the readability and clarity of the definition, as the application of the composition operator ⊗ to more than two components results in highly complicated signature.

We continue by showing that the composition, `compSPS`, adheres to all safety and security constraints of the overall system. This means that the system requirements are satisfied by the composite of the subsystems. This is encoded as the following lemma. We fix an SPF of appropriate signature and assume it to be one of the functions contained in the set of functions describing the composition. We further assume (`and`) the names `getRequest`, `trafficData`, and `getResponse` to refer to the appropriate streams by employing the bundles getter functions. We then show all four safety and the security constraint to hold:

```
lemma compSPS_def_SysReq:
     fixes f :: "InSys^Ω ⇒w OutSys^Ω"
  assumes "f ∈ compSPS speedLimit publicPlates"
      and "getRequest = (fst (bundleInSys.sbgetter·input))"
      and "trafficData = (snd (bundleInSys.sbgetter·input))"
      and "getResponse = (bundleOutSys.sbgetter·(f·input))"
    shows "safety_1 trafficData getResponse
         ∧ safety_2 speedLimit getResponse
         ∧ safety_3 getRequest getResponse
         ∧ safety_4 publicPlates getRequest getResponse
         ∧ security_1 publicPlates trafficData getRequest
             getResponse"
```

This lemma is ultimately implied by the causation between behavioral predicates of system requirements and each individual component. Specifically, we have shown that the safety and security constraints can be implied from a conjunction of all safety and security constraints of the three subsystems, speed camera, database, and API. Despite the proof sketch being conceptually easy, technicalities such as low-level details and intermediate lemmas render the actual proofs in Isabelle more complicated and hard to follow. Thus, we do not present those proofs here in detail and instead refer the interested reader to the appendix.

Finally, we show that `compSPS` is a subset of `SysReqSPS`, which implies that the specification composite of the three components, namely speed camera, database, and user interface, is a valid refinement of the system requirements specification.

```
theorem correct_refinement: "(compSPS speedLimit publicPlates)
                           ⊆ (SysReqSPS speedLimit publicPlates)"
```

Thus, we are safe to conclude that the composition of the given components is correctly designed to fulfill the expectation.

Chapter 7

Conclusion

As highlighted in the introduction, the consideration of time during the specification and verification of real-time systems is crucial. These systems are often designed to operate in environments where the timing of events is critical, and even slight deviations from the expected timing can have catastrophic consequences. It is essential to precisely comprehend the temporal behavior of such systems, emphasizing the need for formal specification and verification methods that take timing into account. This work focuses on extending our verification framework for modeling component networks by incorporating time-based modelization and verification. We have extended existing implementations to incorporate various timed concepts in Isabelle by introducing new datatypes and functions based on related concepts. In the second chapter preliminaries of domain theory and of Isabelle were presented. Alternative formalisms as well as further theorem provers were discussed. The third chapter introduced the timed streams as well as their implementations in Isabelle. Based on timed streams, the fourth chapter focused on the timed bundles and functions over bundles. In the fifth chapter we presented deterministic and non-deterministic timed stream processing functions and specification, including realizability, composition and refinement. The sixth chapter presented a running example of a distributed speed-monitoring system to demonstrate the capabilities of our extended verification framework. We used a relevant property describing one of the temporal behaviors of the system to showcase the capability of our new timed framework to specify and verify timed systems.

Despite the fact that taking timed concepts into account empowered our verification framework significantly, this book presented mainly a foundational layer for timed specifications. It is extremely powerful, but also comes at a cost: It is rather difficult to use. Therefore, plenty of comfort techniques are needed. We, e.g., plan to incorporate automata theory into the verification environment in future publications. Specifically, we plan to focus on the study of event-driven input/output automata, where timing plays a critical role. In order to cover further aspects of our specification and verification methodologies, we will work on bringing up a running example focusing on more complicated industrial use cases, which can also help identify any unnoticed limitations or potential improvements of the framework. Apart from that, we will also consider deepening the integration of model-driven theory generators [Rum16, Rum17]

converting models in architecture description languages (e.g., MontiArc [Wor21, Hab16] or SysML [JPR+22, KPRR21, DRW+20]) to theories in Isabelle with our verification environment, which helps simplify the system specification process and provides a more user-friendly front-end.

Acknowledgment Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 499241390 (FeCoMASS) and German Federal Ministry for Economic Affairs and Climate Action, AMoBaCoD-Project (Grant No. 20X2201C).

Bibliography

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. Deductive Software Verification - The KeY Book. *Lecture notes in computer science*, 10001, 2016.

[AJ95] Samson Abramsky and Achim Jung. *Domain Theory*, page 1–168. Oxford University Press, Inc., USA, 1995.

[AL94] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, 1994.

[AL95] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, 1995.

[ALRL04] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[BC10] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[BC11] Manfred Broy and María Victoria Cengarle. Uml formal semantics: lessons learned. *Software and Systems Modeling*, 10:441–446, 2011.

[BCD+06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 364–387, Springer, Berlin, Heidelberg, 2006.

[BCG12] Manfred Broy, María Victoria Cengarle, and Eva Geisberger. Cyber-physical systems: Imminent challenges. In Radu Calinescu and David Garlan, editors, *Large-Scale Complex IT Systems. Development, Operation and Management*, pages 1–28, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[BCM20] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising Perfectoid Spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, CPP 2020, page 299–312, Association for Computing Machinery, New York, NY, USA, 2020.

[BFH+10] Manfred Broy, Martin Feilkas, Markus Herrmannsdoerfer, Stefano Merenda, and Daniel Ratiu. Seamless model-based development: From isolated tools to integrated model engineering environments. *Proceedings of the IEEE*, 98(4):526–545, 2010.

[BGL+15] David Broman, Lev Greenberg, Edward A. Lee, Michael Masin, Stavros Tripakis, and Michael Wetter. Requirements for Hybrid Cosimulation Standards. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, page 179–188, Association for Computing Machinery, New York, NY, USA, 2015.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BHS99] Manfred Broy, Franz Huber, and Bernhard Schätz. AutoFocus–Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Informatik Forschung und Entwicklung*, 14(3):121–134, 1999.

[Bir07] Alessandro Birolini. *Reliability engineering: Theory and Practice*. Springer, Berlin, 2007.

[BKM07] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Transactions of Software Engineering and Methodology*, 16(1), February 2007.

[BKP+25] Sebastian Bindick, Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Amelie Rath, Bernhard Rumpe, and Andreas Schweiger. Communication and Architectural Patterns for Developing Distributed Systems. In *8th International Conference on Software and Systems Engineering (ICoSSE)*, page 8, IEEE, Nice, France, April 2025.

[BKPS07] Manfred Broy, Ingolf H. Kruger, Alexander Pretschner, and Christian Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[BKR+20] Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla. *Towards an Isabelle Theory for distributed, interactive systems - the untimed case.* Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, March 2020.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BÖFT09] Kyungmin Bae, Peter Csaba Ölveczky, Thomas Huining Feng, and Stavros Tripakis. Verifying Ptolemy II Discrete-Event Models Using Real-Time Maude. In Karin Breitman and Ana Cavalcanti, editors, *Formal methods and software engineering*, Lecture Notes in Computer Science, pages 717–736, Springer, Berlin, 2009.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR23] Manfred Broy and Bernhard Rumpe. Development Use Cases for Semantics-Driven Modeling Languages. *Communications of the ACM*, 66(5):62–71, May 2023.

[Bro86] Manfred Broy. A theory for nondeterminism, parallelism, communication, and concurrency. *Theoretical Computer Science*, 45:1–61, 1986.

[Bro87] Manfred Broy. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing*, 2:13–31, 1987.

[Bro88] Manfred Broy. Nondeterministic data flow programs: How to avoid the merge anomaly. *Science of Computer Programming*, 10(1):65–85, 1988.

[Bro91] Manfred Broy. Towards a formal foundation of the specification and description language sdl. *Form. Asp. Comput.*, 3(1):21–57, January 1991.

[Bro93a] Manfred Broy. Functional specification of time-sensitive communicating systems. *ACM Trans. Softw. Eng. Methodol.*, 2(1):1–46, January 1993.

[Bro93b] Manfred Broy. (Inter-)Action Refinement: The Easy Way. In Manfred Broy, editor, *Program Design Calculi*, pages 121–158, Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[Bro97a] Manfred Broy. Compositional refinement of interactive systems. *J. ACM*, 44(6):850–891, November 1997.

[Bro97b] Manfred Broy. Refinement of time. In Miquel Bertran and Teodor Rus, editors, *Transformation-Based Reactive Systems Development*, pages 44–63, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[Bro06a] M. Broy. The 'grand challenge' in informatics: Engineering software-intensive systems. *Computer*, 39(10):72–80, 2006.

[Bro06b] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 33–42, Association for Computing Machinery, New York, NY, USA, 2006.

[Bro10a] Manfred Broy. A logical basis for component-oriented software and systems engineering. *The Computer Journal*, 53(10):1758–1782, 2010.

[Bro10b] Manfred Broy. Multifunctional software systems: Structured modeling and specification of functional requirements. *Science of Computer Programming*, 75(12):1193–1214, 2010.

[Bro12] Manfred Broy. System behaviour models with discrete and dense time. In Samarjit Chakraborty and Jörg Eberspächer, editors, *Advances in Real-Time Systems*, pages 3–25. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Bro13] Manfred Broy. Engineering cyber-physical systems: Challenges and foundations. In Marc Aiguier, Yves Caseau, Daniel Krob, and Antoine Rauzy, editors, *Complex Systems Design & Management*, pages 1–13, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Bro14] Manfred Broy. A model of dynamic systems. In Saddek Bensalem, Yassine Lakhneck, and Axel Legay, editors, *From Programs to Systems. The Systems perspective in Computing*, pages 39–53. Springer Berlin Heidelberg, 2014.

[Bro23a] Manfred Broy. *Logische und Methodische Grundlagen der Entwicklung verteilter Systeme*. Springer Vieweg Berlin, Heidelberg, 2023.

[Bro23b] Manfred Broy. Specification and verification of concurrent systems by causality and realizability. *Theoretical Computer Science*, 974, 2023.

[Bro24] Manfred Broy. A Calculus for the Specification, Design, and Verification of Distributed Concurrent Systems. *Formal Aspects of Computing*, 36(3), September 2024.

[BŞ01a] Manfred Broy and Gheorghe Ştefănescu. The algebra of stream processing functions. *Theoretical Computer Science*, 258(1):99–129, 2001.

[BS01b] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement.* Springer Verlag Heidelberg, 2001.

[BS14] Manfred Broy and Albrecht Schmidt. Challenges in engineering cyber-physical systems. *Computer*, 47(2):70–72, 2014.

[BS24] Manfred Broy and Bran Selić. Specifying and Composing Layered Architectures. *Journal of Object Technology*, 23(1):1:1–24, 2024.

[CDL+12] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA+ proofs. In *FM 2012: Formal Methods: 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings 18*, pages 147–154. Springer, 2012.

[CFL08] Chihhong Patrick Cheng, Teale Fristoe, and Edward A Lee. Applied Verification: The Ptolemy Approach. Technical Report UCB/EECS-2008-41, EECS Department, University of California, Berkeley, Apr 2008.

[CKM+99] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. Defining UML Family Members Using Prefaces. In Ch. Mingins and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems, TOOLS'99 Pacific*, IEEE Computer Society, 1999.

[Cla14] Claudius Ptolemaeus. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014.

[CM14] Darren Cofer and Steven P Miller. Formal methods case studies for DO-333. Technical Report NF1676L-18435, Langley Research Center, 2014.

[DFL+08] Patricia Derler, Thomas Huining Feng, Edward A. Lee, Slobodan Matic, Hiren D. Patel, Yang Zhao, and Jia Zou. PTIDES: A Programming Model for Distributed Real-Time Embedded Systems. Technical Report UCB/EECS-2008-72, EECS Department, University of California, Berkeley, May 2008.

[DM07] Gilles Dowek and César Muñoz. Conflict Detection and Resolution for 1,2,... N Aircraft. In *Proceedings of the 6th AIAA Aviation Technology, Integration and Operations Conference (ATIO)*, Belfast, Northern Ireland, September 2007.

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Springer International Publishing, Cham, 2015.

[DRW+20] Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. Modeling Mechanical Functional Architectures in SysML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 79–89. ACM, October 2020.

[EBF+98] Andy Evans, Jean-Michel Bruel, Robert France, Kevin Lano, and Bernhard Rumpe. Making UML Precise. In *OOPSLA'98 Workshop on "Formalizing UML. Why and How?"*, Vancouver, Canada, October 1998.

[FKR+25] Lars Fischer, Hendrik Kausch, Bernhard Rumpe, Max Stachon, Sebastian Stüber, and Lucas Wollenhaupt. Methodical and Formally Verified Model-Driven Architecture Refactoring. *Journal of Object Technology (JOT)*, 24(2):1–14, May 2025.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0: Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKRB96] Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy. State Transition Diagrams. Technical report, TU Munich, 1996.

[GR06] Borislav Gajanovic and Bernhard Rumpe. Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen. Technical report, TU Braunschweig, Carl-Friedrich-Gauss-Fakultät für Mathematik und Informatik, 2006.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems.* Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[Hen96] T.A. Henzinger. The theory of hybrid automata. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 278–292, 1996.

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021.* Aachener

Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HLMSN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, August 1978.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal*, 37(10):64–72, October 2004.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. *MontiArc - Architectural modeling of interactive distributed and cyber-physical systems*, Technical report / Department of Computer Science, RWTH Aachen 2012,3. RWTH and Technische Informationsbibliothek u. Universitätsbibliothek and Niedersächische Staats- und Universitätsbibliothek, Aachen and Hannover and Göttingen, 2012.

[HST10] F. Hölzl, M. Spichkova, and D. Trachtenherz. Autofocus tool chain. Technical report, Technical University Munich, 2010.

[Huf12] Brian Charles Huffman. *HOLCF'11: A Definitional Domain Theory for Verifying Functional Programs*. PhD thesis, Portland State University, 2012.

[JPR+22] Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. The Language of SysML v2 under the Magnifying Glass. *Journal of Object Technology (JOT)*, 21:1–15, July 2022.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood.

seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, Association for Computing Machinery, New York, NY, USA, 2009.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Springer Verlag, Berlin, 1999.

[KKN+24] Hendrik Kausch, Kristiaan Koppes, Lukas Netz, Patrick O'Brien, Mathias Pfeiffer, Deni Raco, Matthias Radny, Amelie Rath, Rebecca Richstein, and Bernhard Rumpe. Applied Model-Based Co-Development for Zero-Emission Flight Systems Based on SysML. In *Deutscher Luft- und Raumfahrtkongress (DLRK 2024)*. Deutsche Gesellschaft für Luft- und Raumfahrt - Lilienthal-Oberth e.V., October 2024.

[Kle52] Stephen Cole Kleene. *Introduction to metamathematics*. van Nostrand New York, 1952.

[KMP+21] Hendrik Kausch, Judith Michael, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021.

[KOB+17] Stefan Kugele, Philipp Obergfell, Manfred Broy, Oliver Creighton, Matthias Traub, and Wolfgang Hopfensitz. On service-orientation for automotive software. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 193–202, 2017.

[KPR+22] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Correct and Sustainable Development Using Model-based Engineering and Formal Methods. In *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*. IEEE, September 2022.

[KPR+23] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Amelie Rath, Bernhard Rumpe, and Andreas Schweiger. A Theory for Event-Driven Specifications Using Focus and MontiArc on the Example of a Data Link Uplink Feed System. In Iris Groher and Thomas Vogel, editors, *Software Engineering 2023 Workshops*, pages 169–188. Gesellschaft für Informatik e.V., February 2023.

[KPR+24] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Enhancing System-model Quality: Evaluation of the MontiBelle Approach with the Avionics Case Study on a Data Link Uplink Feed System. In *Avionics Systems and Software Engineering Workshop of the Software Engineering 2024 - Companion Proceedings (AvioSE)*, pages 119–138. Gesellschaft für Informatik e.V., Februrary 2024.

[KPR+25a] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Enhancing System Model Quality: Evaluation of the Systems Modeling Language (SysML)-Driven Approach in Avionics. *Journal of Aerospace Information Systems (JAIS)*, 22(5):367–378, May 2025.

[KPR+25b] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas 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(1):33–48, January 2025.

[KPRR20a] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, and Bernhard Rumpe. An Approach for Logic-based Knowledge Representation and Automated Reasoning over Underspecification and Refinement in Safety-Critical Cyber-Physical Systems. In Regina Hebig and Robert Heinrich, editors, *Combined Proceedings of the Workshops at Software Engineering 2020*. CEUR Workshop Proceedings, February 2020.

[KPRR20b] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, and Bernhard Rumpe. MontiBelle - Toolbox for a Model-Based Development and Verification of Distributed Critical Systems for Compliance with Functional Safety. In *AIAA Scitech 2020 Forum*. American Institute of Aeronautics and Astronautics, January 2020.

[KPRR21] Hendrik Kausch, Mathias Pfeiffer, Deni Raco, and Bernhard Rumpe. Model-Based Design of Correct Safety-Critical Systems using Dataflow Languages on the Example of SysML Architecture and Behavior Diagrams. In Sebastian Götz, Lukas Linsbauer, Ina Schaefer, and Andreas Wortmann, editors, *Proceedings of the Software Engineering 2021 Satellite Events*. CEUR, February 2021.

[KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung,

Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 05 1994.

[Lam99] Leslie Lamport. Specifying Concurrent Systems with TLA+. *Calculational System Design*, pages 183–247, April 1999.

[Lee17] Edward A. Lee. Fundamental Limits of Cyber-Physical Systems Modeling. *ACM Transactions on Cyber-Physical Systems*, 1(1):1–26, 2017.

[Lei10] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Ler09] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, 06 2009.

[LF08] Michael Leuschel and Marc Fontaine. Probing the Depths of CSP-M: A New FDR-Compliant Validation Tool. In Shaoying Liu, Tom Maibaum, and Keijiro Araki, editors, *Formal Methods and Software Engineering*, pages 278–297, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[LSVW96] Nancy Lynch, Roberto Segala, Frits Vaandrager, and H. B. Weinberg. Hybrid I/O automata. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, pages 496–510, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[Lyn89] N. Lynch. Multivalued possibilities mappings. *In: J. W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): Stepwise Refinement of Distributed Systems.*, 1989.

[MAD+19] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin HriŢcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In Luís Caires, editor, *Programming Languages and Systems*, pages 30–59, Springer International Publishing, Cham, 2019.

[Mea55] George H. Mealy. A Method for Synthesizing Sequential Circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer Berlin, Heidelberg, 1980.

[Mil99] Robin Milner. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press, 1999.

[MNvOS99] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[Neu12] Philipp Neubeck. *A Probabilitistic Theory of Interactive Systems*. PhD thesis, Technische Universität München, 2012.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Lecture notes in Artificial Intelligence 2283. Springer, Berlin, 2002.

[ÖM07] Peter Csaba Ölveczky and José Meseguer. Semantics and Pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Automated Deduction—CADE-11*, pages 748–752, Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.

[Ost89] J. S. Ostroff. Automated verification of timed transition models. *J. Sifakis (ed.): AutomaticVerification Methods for Finite State Systems*, 1989.

[PBMB19] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally Verified Cryptographic Web Applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1256–1274, 2019.

[PCB+01] Pamela Paxton, Patrick J. Curran, Kenneth A. Bollen, Jim Kirby, and Feinian Chen. Monte Carlo Experiments: Design and Implementation.

Structural Equation Modeling: A Multidisciplinary Journal, 8(2):287–312, 2001.

[PHDB16] Klaus Pohl, Harald Hönninger, Heinrich Daembkes, and Manfred Broy, editors. *Advanced Model-Based Engineering of Embedded Systems*. Springer, Cham, 2016.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR97] Jan Philipps and Bernhard Rumpe. Refinement of Information Flow Architectures. In M. Hinchey, editor, *ICFEM'97 Proceedings*, IEEE CS Press, Hiroshima, Japan, 1997.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[RDLF+19] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified secure Zero-Copy parsers for authenticated message formats. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1465–1482, 2019.

[Reg94] Franz Regensburger. *HOLCF: eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technical University Munich, Germany, 1994.

[Rei95] Wolfgang Reif. The Kiv-Approach to Software Verification. In Manfred Broy and Stefan Jähnichen, editors, *KORSO: Methods, Languages, and Tools for the Construction of Correct Software: Final Report*, pages 339–368. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall PTR, USA, 1997.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum98] Bernhard Rumpe. A Note on Semantics (with an Emphasis on UML). In Haim Kilov and Bernhard Rumpe, editors, *Second ECOOP Workshop on Precise Behavioral Semantics*, TUM-I9813, 1998.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods.* Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring.* Springer International, May 2017.

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.

[SBBR22] Gerhard Schellhorn, Stefan Bodenmüller, Martin Bitterlich, and Wolfgang Reif. Software & System Verification with KIV. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, and Einar Broch Johnsen, editors, *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*, pages 408–436. Springer International Publishing, Cham, 2022.

[SCF+11] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure Distributed Programming with Value-Dependent Types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 266–278, Association for Computing Machinery, New York, NY, USA, 2011.

[Sch22] Peter Scholze. Liquid Tensor Experiment. *Experimental Mathematics*, 31(2):349–354, 2022.

[Sha49] C. E. Shannon. Communication in the Presence of Noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[SHLG94] Viggo Stoltenberg-Hansen, Ingrid Lindstroem, and Edward R Griffor. *Mathematical Theory of Domains*. Cambridge University Press, 1994.

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[SLD08] Jun Sun, Yang Liu, and Jin Song Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation*, pages 307–322, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[SMR23] Nikhil Swamy, Guido Martínez, and Aseem Rastogi. *Proof-Oriented Programming in F**, Janurary 2023.

[Spi07] Maria Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. Dissertation, Technische Universität München, München, 2007.

[SRS99] Thomas Stauner, Bernhard Rumpe, and Peter Scholz. Hybrid System Model. Technical report, TU Munich, 1999.

[Tar55] Alfred Tarski. A Lattice-Theoretical Fix Point Theorem and Its Applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955.

[Wen02] Markus Wenzel. *Isabelle, Isar - a versatile environment for human readable formal proof documents*. PhD thesis, Technical University Munich, Germany, 2002.

[Wor21] Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 50. Shaker Verlag, October 2021.

[ZBPB17] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, CCS '17, page 1789–1806, Association for Computing Machinery, New York, NY, USA, 2017.

List of Figures

List of Tables

Glossary

Application Programming Interfaces (API) Common interface for communication between independent applications. 96, 97, 99

Architecture Description Language (ADL) Language to specify and model the structure and components of software architectures. 58

Bottom Least element in a pointed complete partial order; denoted as $\bot$. 21, 28, 29

Causality Property of timed components. Outputs rely solely on past inputs. 35, 73, 74, 76–79, 85

Chain Totally ordered set with minimal element. 74, 75

Channel Asynchronous, point-to-many, instantaneous message-passing method for communication between components. 27, 29, 78

complete partial order (cpo) A partial order in which every chain has a least upper bound. 13, 14, 18, 20, 52, 55

Component Building block for systems with an explicit communication interface. 27, 35, 73, 77, 78

Composition Combine components into larger systems. 78

Continuous The least upper bound is preserved after application of the function. 21, 34, 38, 39, 42, 73–75, 77, 85

Deterministic Behavior is fully determined by input and current state. 73, 85

Event Either a message or a Tick that marks the completion of a time slice. 29–32, 34, 35, 37, 40–42, 51, 52, 81

Feedback Composition Special kind of composition. Output channels are used as input of the same component. 80

Fixpoint Solution x for a function f where $f(x) = x$. 92

greatest fixpoint (gfp) Used to define the semantic of recursive definitions. 92

Higher Order Logic of Computable Functions (HOLCF) Isabelle Library for Continuous Functions. 18, 19, 21, 29

Isar A proof language in Isabelle. Designed to be similar to handwritten proofs. 22

lazy natural number (lNat) Natural numbers extended with infinity. 37, 38

least fixed point (lfp) Used to define the semantic of recursive definitions. 14, 21, 87, 92

least upper bound (lub) The least element in the set of upper bounds. 13–15

Monotonic The order is preserved after application of the function. 13, 37, 74, 75, 77

MontiArc Framework for modeling and simulation of software architectures. 58

Non-Deterministic Behavior is not completely determined by input and current state. 73

partial order (po) A reflexive, transitive and antisymmetric relation. 12–14, 18

pointed complete partial order (pcpo) A complete partial order with a bottom element. 13, 14, 18, 21, 28, 29, 34, 52, 61

Prefix Order A stream is considered lesser if it serves as a prefix to another stream. 28, 52, 75

Sequential Composition Special kind of composition. The output of the first component is the input of the second component. 79, 80

stream bundle (SB) Combination of multiple streams each identified by a channel. 3, 8, 34, 57–64, 68–71, 98

stream processing specification (SPS) Set of stream-processing functions, used to described non-deterministic behavior. 3, 8, 87, 88, 90, 92, 101, 109

stream-processing function (SPF) Continuous function mapping input bundle to output bundle. 3, 8, 81, 85, 87, 89, 90, 93, 101, 109

Strong Causality Property of timed components. Outputs rely solely on past inputs and introduce delays. 34, 73–78, 80

SysML SysML is a general-purpose modeling language for modeling complex systems both textually and graphically. 58

Time Slice Represents a specific interval of time during which measurements or observations occur, often marked by a signal indicating its end. 30–33, 35, 37–40, 42, 43, 47, 48, 52, 73–76, 79, 81, 82, 86

Time Synchronous At most one message per time slice, which can either contain a message or be empty. 31, 33–35, 37, 39, 46

timed stream bundle (TSB) Combination of multiple timed streams each identified by a channel. 34, 35, 57, 58, 60–62, 64–72

timed stream processing function (TSPF) Continuous, Deterministic Function over timed input and output. 77, 78, 81–85, 87

timed stream processing specification (TSPS) Set of timed stream-processing functions, used to described non-deterministic behavior. 87, 90

TOne Exactly one message per time slice. 32–35, 37, 49

Untimed No timing information for messages, focusing solely on the content without temporal context. 29, 35, 36, 38, 40, 42, 73

Weak Causality Property of timed components. Outputs rely solely on past inputs. 34, 73–75, 77–79, 81, 85

Wellformedness Restrict timed observation by forbidding infinite messages during a finite duration. 30, 35, 39, 51, 67, 75, 78

Related Work from the SE Group, RWTH Aachen, June 25

The following section gives an overview of related work done at the SE Group, RWTH Aachen. More details can be found on the website `www.se-rwth.de/topics/` or in [HMR+19]. The work presented here mainly has been guided by our mission statement:

Our mission is to define, improve, and industrially apply *techniques*, *concepts*, and *methods* for *innovative and efficient development* of software and software-intensive systems, such that *high-quality* products can be developed in a *shorter period of time* and with *flexible integration* of *changing requirements*. Furthermore, we *demonstrate the applicability* of our results in various domains and potentially refine these results in a domain specific form.

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04c]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process.", [JWCR18] addresses the question of how digital and organizational techniques help to cope with the physical distance of developers and [RRSW17] addresses how to teach agile modeling.

Modeling will increasingly be used in development projects if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKR+06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum11, Rum12] and [Rum16, Rum17], the UML/P, a variant of the UML especially designed for programming, refactoring, and evolution is defined.

The language workbench MontiCore [GKR+06, GKR+08, HKR21] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09], and refactoring in various modeling and programming languages [PR03]. To better understand the effect of an agile evolving design, we discuss the need for semantic differencing in [MRR10].

In [FHR08] we describe a set of general requirements for model quality. Finally, [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve the reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation. In [KMA+16] we have also introduced a classification of ways to reuse modeled software components.

Artifacts in Complex Development Projects

Developing modern software solutions has become an increasingly complex and time consuming process. Managing the complexity, the size, and the number of artifacts developed and used during a project together with their complex relationships is not trivial [BGRW17].

To keep track of relevant structures, artifacts, and their relations in order to be able, e.g., to evolve or adapt models and their implementing code, the *artifact model* [GHR17, Gre19] was introduced. [BGRW18] and [HJK+21] explain its applicability in systems engineering based on MDSE projects and [BHR+18] applies a variant of the artifact model to evolutionary development, especially for CPS.

An artifact model is a meta-data structure that explains which kinds of artifacts, namely code files, models, requirements files, etc. exist and how these artifacts are related to each other. The artifact model, therefore, covers the wide range of human activities during the development down to fully automated, repeatable build scripts. The artifact model can be used to optimize parallelization during the development and building, but also to identify deviations of the real architecture and dependencies from the desired, idealistic architecture, for cost estimations, for requirements and bug tracing, etc. Results can be measured using metrics or visualized as graphs.

Artificial Intelligence in Software Engineering

MontiAnna is a family of explicit domain specific languages for the concise description of the architecture of (1) a neural network, (2) its training, and (3) the training data [KNP+19]. We have developed a compositional technique to integrate neural networks into larger software architectures [KRRW17] as standardized machine learning components [KPRS19]. This enables the compiler to support the systems engineer by automating the lifecycle of such components including multiple learning approaches such as supervised learning, reinforcement learning, or generative adversarial networks.

For analysis of MLOps in an agile development, a software 2.0 artifact model distinguishing different kinds of artifacts is given in [AKK+21].

According to [MRR11g] the semantic difference between two models are the elements contained in the semantics of the one model that are not elements in the semantics of the other model. A smart semantic differencing operator is an automatic procedure for computing diff witnesses for two given models. Such operators have been defined for Activity Diagrams [MRR11d], Class Diagrams [MRR11b], Feature Models [DKMR19], Statecharts [DEKR19], and Message-Driven Component and Connector Architectures [BKRW17, BKRW19]. We also developed a modeling language-independent method for determining syntactic changes that are responsible for the existence of semantic differences [KR18a].

We apply logic, knowledge representation, and intelligent reasoning to software engineering to perform correctness proofs, execute symbolic tests, or find counterexamples using a theorem prover. We have defined a core theory in [BKR+20], which is based on the core concepts of Broy's Focus theory [RR11, BR07], and applied it to challenges in intelligent flight control systems and assistance systems for air or road traffic management [KRRS19, KMP+21, HRR12].

Intelligent testing strategies have been applied to automotive software engineering [EJK+19, DGH+19, KMS+18], or more generally in systems engineering [DGH+18]. These methods are realized for a variant of SysML Activity Diagrams (ADs) and Statecharts.

Machine Learning has been applied to the massive amount of observable data in energy management for buildings [FLP+11, KLPR12] and city quarters [GLPR15] to optimize operational efficiency and prevent unneeded CO_2 emissions or reduce costs. This creates a structural and behavioral system theoretical view on cyber-physical systems understandable as essential parts of digital twins [RW18, BDH+20].

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivate of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P, [Hab16] for MontiArc is used in domains such as cars or robotics

[HRR12], and [AMN+20a] for enterprise information systems based on the MontiCore language workbench [KRV10, GKR+06, GKR+08, HKR21].

In [KRV06], we discuss additional roles necessary in a model-based software development project. [GKR+06, GHK+15, GHK+15a] discuss mechanisms to keep generated and handwritten code separated. In [Wei12, HRW15, Hoe18], we demonstrate how to systematically derive a transformation language in concrete syntax and, e.g., in [HHR+15, AHRW17] we have applied this technique successfully for several UML sub-languages and DSLs.

[HNRW16] presents how to generate extensible and statically type-safe visitors. In [NRR16], we propose the use of symbols for ensuring the validity of generated source code. [GMR+16] discusses product lines of template-based code generators. We also developed an approach for engineering reusable language components [HLN+15, HLN+15a].

To understand the implications of executability for UML, we discuss the needs and the advantages of executable modeling with UML in agile projects in [Rum04c], how to apply UML for testing in [Rum03], and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML) & the UML-P Tool

Starting with the early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the books [Rum16, Rum17] and is implemented in [Sch12].

Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09], [BCGR09a], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied when checking variants of class diagrams [MRR11e] and object diagrams [MRR11c] or the consistency of both kinds of diagrams [MRR11f]. We also apply these concepts to activity diagrams [MRR11a] which allows us to check for semantic differences in activity diagrams [MRR11d]. The basic semantics for ADs and their semantic variation points are given in [GRR10].

We also discuss how to ensure and identify model quality [FHR08], how models, views, and the system under development correlate to each other [BGH+98b], and how to use modeling in agile development projects [Rum04c], [Rum03] and [Rum02].

The question of how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99a], [FEL+98] and [SRVK10].

The UML-P tool was conceptually defined in [Rum16, Rum17, Rum12, Rum11], got the first realization in [Sch12], and is extended in various ways, such as logically or physically distributed computation [BKRW17a]. Based on a detailed examination [JPR+22], insights are also transferred to the SysML 2.

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use than general-purpose programming languages but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08, HKR21] allows the specification of an integrated abstract and concrete syntax format [KRV07b, HKR21] for easy development. New languages

and tools can be defined in modular forms [KRV08, GKR+07, Voe11, HLN+15, HLN+15a, HRW18, BEK+18b, BEK+19, Sch12] and can, thus, easily be reused. We discuss the roles in software development using domain specific languages already in [KRV06] and elaborate on the engineering aspect of DSL development in [CFJ+16].

[Wei12, HRW15, Hoe18] present an approach that allows the creation of transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11, GMR+16]. [BDL+18] presents a method to derive internal DSLs from grammars. In [BJRW18], we discuss the translation from grammars to accurate metamodels. Successful applications have been carried out in the Air Traffic Management [ZPK+11] and television [DHH+20] domains. Based on the concepts described above, meta modeling, model analyses, and model evolution have been discussed in [LRSS10] and [SRVK10]. [BJRW18] describes a mapping bridge between both. DSL quality in [FHR08], instructions for defining views [GHK+07] and [PFR02], guidelines to define DSLs [KKP+09], and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

A broader discussion on the *global* integration of DSMLs is given in [CBCR15] as part of [CCF+15a], and [TAB+21] discusses the compositionality of analysis techniques for models.

The MontiCore language workbench has been successfully applied to a larger number of domains, resulting in a variety of languages documented, e.g., in [AHRW17, BEH+20, BHR+21, BPR+20, HHR+15, HJRW20, HMR+19, HRR12, PBI+16, RRW15] and Ph.D. theses like [Ber10, Gre19, Hab16, Her19, Kus21, Loo17, Pin14, Plo18, Rei16, Rot17, Sch12, Wor16].

Software Language Engineering

For a systematic definition of languages using a composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15a]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10, HR17, HKR21, HRW18, BPR+20, BEK+19].

In [SRVK10] we discuss the possibilities and the challenges of using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Voe11, Naz17, KRV08, HLN+15, HLN+15a, HNRW16, HKR21, BEK+18b, BEK+19] and the backend [RRRW15b, NRR16, GMR+16, HKR21, BEK+18b, BBC+18]. In [GHK+15, GHK+15a], we discuss the integration of handwritten and generated object-oriented code. [KRV10] describes the roles in software development using domain specific languages.

Language derivation is to our belief a promising technique to develop new languages for a specific purpose, e.g., model transformation, that relies on existing basic languages [HRW18].

How to automatically derive such a transformation language using a concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs.

We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15, HHK+13] that are derived from base languages to be able to constructively describe differences between model variants usable to build feature sets.

The derivation of internal DSLs from grammars is discussed in [BDL+18] and a translation of grammars to accurate metamodels in [BJRW18].

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services.

We use streams, statemachines [GKR+96], and components [BR07] as well as expressive forms of composition and refinement [PR99, PR97, RW18] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR10, HRR12] for architecture design and extensions for states [RRW13c, BKRW17a, RRW14a, Wor16]. In [RRW13], we introduce a code generation framework for MontiArc. [RRRW15b] describes how the language is composed of individual sublanguages.

MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, HKR+11] and evolution on deltas [HRRS12]. Other extensions are concerned with modeling cloud architectures [PR13], security in [HHR+15], and the robotics domain [AHRW17, AHRW17b]. Extension mechanisms for MontiArc are generally discussed in [BHH+17].

[GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants.

[MRR14b] provides a precise technique for verifying the consistency of architectural views [Rin14, MRR13] against a complete architecture to increase reusability. We discuss the synthesis problem for these views in [MRR14a]. An experience report [MRRW16] and a methodological embedding [DGH+19] complete the core approach.

Extensions for co-evolution of architecture are discussed in [MMR10], for powerful analyses of software architecture behavior evolution provided in [BKRW19], techniques for understanding semantic differences presented in [BKRW17], and modeling techniques to describe dynamic architectures shown in [HRR98, HKR+16, BHK+17, KKR19].

Compositionality & Modularity of Models

[HKR+09, TAB+21] motivate the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07, RW18] and algebraically grounded in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10, HKR21] that can even be used to develop modeling tools in a compositional form [HKR21, HLN+15, HLN+15a, HNRW16, NRR16, HRW18, BEK+18b, BEK+19, BPR+20, KRV07b]. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09].

[Voe11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular editor generation is discussed in [KRV07a]. [RRRW15b] applies compositionality to robotics control.

[CBCR15] (published in [CCF+15a]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information, we have developed the concept of tagging languages in [GLRR15, MRRW16]. It allows the description of additional information for model elements in separated documents, facilitates reuse, and allows typing tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision, and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96, RK96]. An extended version especially suited for the UML is given in [GRR09], [BCGR09a] and in [BCGR09] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08] or sequence diagrams in [BGH+98a].

To better understand the effect of an evolved design, detection of semantic differencing, as opposed to pure syntactical differences, is needed [MRR10]. [MRR11d, MRR11a] encode a part of the semantics to handle semantic differences of activity diagrams. [MRR11f, MRR11f] compare class and object diagrams with regard to their semantics. Furthermore, [BKRW17] compares component and connector architectures similar to SysML' block definition diagrams and [RSR+99] discusses the combination of those architectures with the UML.

In [BR07, RR11], a precise mathematical model for distributed systems based on black-box behaviors of components is defined and accompanied by automata in [Rum96]. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of exemplary object interaction, today called sequence diagram. [BGH+98b] discusses the relationships between a system, a view, and a complete model in the context of the UML.

[GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these to class and object diagrams in [MRR11f] as well as activity diagrams in [GRR10].

[Rum12] defines the semantics in a variety of code and test case generation, refactoring, and evolution techniques. [LRSS10] discusses the evolution and related issues in greater detail. [RW18] discusses an elaborated theory for the modeling of underspecification, hierarchical composition, and refinement that can be practically applied to the development of CPS.

A first encoding of these theories in the Isabelle verification tool is defined in [BKR+20].

Evolution and Transformation of Models

Models are the central artifacts in model driven development, but as code, they are not initially correct and need to be changed, evolved, and maintained over time. Model transformation is therefore essential to effectively deal with models [CFJ+16].

Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04c, MRR10], refinement [PR99, PR97, KPR97, PR94], decomposition [PR99, KRW20], synthesis [MRR14a], refactoring [Rum12, PR03], translating models from one language into another [MRR11e, Rum12], systematic model transformation language development [Wei12, HRW15, Hoe18, HHR+15], repair of failed model evolution [KR18a].

[Rum04c] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97] and refining pipe-and-filter architectures is explained in [PR99, PR97]. This has e.g. been applied for robotics in [AHRW17, AHRW17b].

Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. [HRRS11, HRR+11, HRRS12] encode these in constructive Delta transformations, which are defined in derivable Delta languages [HHK+13].

Translation between languages, e.g., from class diagrams into Alloy [MRR11e] allows for comparing class diagrams on a semantic level. Similarly, semantic differences of evolved activity diagrams are identified via techniques from [MRR11d] and for Simulink models in [RSW+15].

Variability and Software Product Lines (SPL)

Products often exist in various variants, for example, cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08, GKPR08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12].

Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRRS11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe special variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13, HHK+15] and [HRW15] describe an approach to systematically derive delta languages.

We also apply variability modeling languages to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02] and generators [GMR+16]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09], leverage features for their compositional reuse [BEK+18b, BEK+19], and applied it as a semantic language refinement on Statecharts in [GR11].

Digital Twins and Digital Shadows in Engineering and Production

The digital transformation of production changes the life cycle of the design, the production, and the use of products [BDJ+22]. To support this transformation, we can use Digital Twins (DTs) and Digital Shadows (DSs). In [DMR+20] we define: "A digital twin of a system consists of a set of models of the system, a set of digital shadows, and provides a set of services to use the data and models purposefully with respect to the original system."

We have investigated how to synthesize self-adaptive DT architectures with model-driven methods [BBD+21a] and have applied it e.g. on a digital twin for injection molding [BDH+20]. In [BDR+21] we investigate the economic implications of digital twin services.

Digital twins also need user interaction and visualization, why we have extended the infrastructure by generating DT cockpits [DMR+20]. To support the DevOps approach in DT engineering, we have created a generator for low-code development platforms for digital twins [DHM+22] and sophisticated tool chains to generate process-aware digital twin cockpits that also include condensed forms of event logs [BMR+22].

[BBD+21b] describes a conceptual model for digital shadows covering the purpose, relevant assets, data, and metadata as well as connections to engineering models. These can be used during the runtime of a DT, e.g. when using process prediction services within DTs [BHK+21].

Integration challenges for digital twin systems-of-systems [MPRW22] include, e.g., the horizontal integration of digital twin parts, the composition of DTs for different perspectives, or how to handle different lifecycle representations of the original system.

Modeling for Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12, BBR20] are complex, distributed systems that control physical entities. In [RW18], we discuss how an elaborated theory can be practically applied to the development of CPS. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12, KRRW17], autonomous driving [BR12b, KKR19], and digital twin development [BDH+20] to processes and tools to improve the development as well as the product itself [BBR07].

In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest to European avionics [ZPK+11]. Optimized [KRS+18a] and domain specific code generation [AHRW17b], and the extension to product lines of CPS [RSW+15, KRR+16, RRS+16] are key for CPS.

A component and connector architecture description language (ADL) suitable for the specific challenges in robotics is discussed in [RRW13c, RRW14a, Wor16, RRSW17, Wor21]. In [RRW12], we use this language for describing requirements and in [RRW13], we describe a code generation framework for this language. Monitoring for smart and energy efficient buildings is developed as an Energy Navigator toolset [KPR12, FPPR12, KLPR12].

Model-Driven Systems Engineering (MDSysE)

Applying models during Systems Engineering activities is based on the long tradition of contributing to systems engineering in automotive [FND+98] and [GHK+08a], which culminated in a new comprehensive model-driven development process for automotive software [KMS+18, DGH+19]. We leveraged SysML to enable the integrated flow from requirements to implementation to integration.

To facilitate the modeling of products, resources, and processes in the context of Industry 4.0, we also conceived a multi-level framework for production engineering based on these concepts [BKL+18] and addressed to bridge the gap between functions and the physical product architecture by modeling mechanical functional architectures in SysML [DRW+20]. For that purpose, we also did a detailed examination of the upcoming SysML 2.0 standard [JPR+22] and examined how to extend the SPES/CrEST methodology for a systems engineering approach [BBR20].

Research within the excellence cluster Internet of Production considers fast decision making at production time with low latencies using contextual data traces of production systems, also known as Digital Shadows (DS) [SHH+20]. We have investigated how to derive Digital Twins (DTs) for injection molding [BDH+20], how to generate interfaces between a cyber-physical system and its DT [KMR+20], and have proposed model-driven architectures for DT cockpit engineering [DMR+20].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using

statemachines resp. Statecharts [GKR96, GKR+96, BCR07b, BCGR09a, BCGR09], (2) understanding the refinement [PR94, RK96, Rum96, RW18] and composition [GR95, GKR96, RW18] of statemachines, and (3) applying statemachines for modeling systems.

In [Rum96, RW18] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [GKR96, BR07].

We apply these techniques, e.g., in MontiArcAutomaton [RRW13, RRW14a, RRW13, RW18], in a robot task modeling language [THR+13], and in building management systems [FLP+11b].

Model-Based Assistance and Information Services (MBAIS)

Assistive systems are a special type of information system: they (1) provide situational support for human behavior (2) based on information from previously stored and real-time monitored structural context and behavior data (3) at the time the person needs or asks for it [HMR+19]. To create them, we follow a model centered architecture approach [MMR+17] which defines systems as a compound of various connected models. Used languages for their definition include DSLs for behavior and structure such as the human cognitive modeling language [MM13], goal modeling languages [MRV20, MRZ21] or UML/P based languages [MNRV19]. [MM15] describes a process of how languages for assistive systems can be created. MontiGem [AMN+20a] is used as the underlying generator technology.

We have designed a system included in a sensor floor able to monitor elderlies and analyze impact patterns for emergency events [LMK+11]. We have investigated the modeling of human contexts for the active assisted living and smart home domain [MS17] and user-centered privacy-driven systems in the IoT domain in combination with process mining systems [MKM+19], differential privacy on event logs of handling and treatment of patients at a hospital [MKB+19], the mark-up of online manuals for devices [SM18a] and websites [SM20], and solutions for privacy-aware environments for cloud services [ELR+17] and in IoT manufacturing [MNRV19]. The user-centered view of the system design allows to track who does what, when, why, where, and how with personal data, makes information about it available via information services and provides support using assistive services.

Modeling Robotics Architectures and Tasks

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires the composition and the interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers the broad propagation of robotics applications.

The MontiArcAutomaton language [RRW12, RRW14a] extends the ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13c, RRRW15b, HKR21] that perfectly fit robotic architectural modeling.

The iserveU modeling framework describes domains, actors, goals, and tasks of service robotics applications [ABH+16, ABH+17] with declarative models. Goals and tasks are translated into models of the planning domain definition language (PDDL) and then solved [ABK+17]. Thus, domain experts focus on describing the domain and its properties only.

The LightRocks [THR+13, BRS+15] framework allows robotics experts and laymen to model robotic assembly tasks. In [AHRW17, AHRW17b], we define a modular architecture modeling method for translating architecture models into modules compatible with different robotics middleware platforms.

Many of the concepts in robotics were derived from automotive software [BBR07, BR12b].

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment, and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs to be managed, developed, and tested. A consistent requirement management connecting requirements with features in all development phases for the automotive domain is described in [GRJA12].

The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. A methodical embedding of the resulting function nets and their quality assurance using automated testing is given in the SMaRDT method [DGH+19, KMS+18].

[HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses the means to extract a well-defined Software Product Line from a set of copy and paste variants.

Potential variants of components in product lines can be identified using similarity analysis of interfaces [KRR+16], or execute tests to identify similar behavior [RRS+16]. [RSW+15] describes an approach to using model checking techniques to identify behavioral differences of Simulink models. In [KKR19], we model dynamic reconfiguration of architectures applied to cooperating vehicles.

Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12b, BR12], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in the development and the evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12b].

[MMR10] gives an overview of the state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions.

MontiSim simulates autonomous and cooperative driving behavior [GKR+17] for testing various forms of errors as well as spatial distance [FIK+18, KKRZ19]. As tooling infrastructure, the SSELab storage, versioning, and management services [HKR12] are essential for many projects.

Internet of Things, Industry 4.0 & the MontiThings Tool

The Internet of Things (IoT) requires the development of increasingly complex distributed systems. The MontiThings ecosystem [KRS+22] provides an end-to-end solution to modeling, deploying [KKR+22], and analyzing [KMR21] failure-tolerant [KRS+22] IoT systems and connecting them to synthesized digital twins [KMR+20]. We have investigated how model-driven methods can support the development of privacy-aware [ELR+17, HHK+14] cloud systems [PR13], distributed systems security [HHR+15], privacy-aware process mining [MKM+19], and distributed robotics applications [RRRW15b].

In the course of Industry 4.0, we have also turned our attention to mechanical or electrical applications [DRW+20]. We identified the digital representation, integration, and (re-)configuration

of automation systems as primary Industry 4.0 concerns [WCB17]. Using a multi-level modeling framework, we support machine as a service approaches [BKL+18].

Smart Energy Management

In the past years, it became more and more evident that saving energy and reducing CO_2 emissions are important challenges. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales.

During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for the technical specification of building services already.

We adapted the well-known concept of statemachines to be able to describe different states of a facility and validate it against the monitored values [FLP+11b]. We show how our data model, the constraint rules, and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing and Services

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality, and new application domains. It promises to enable new business models, facilitate web-based innovations, and increase the efficiency and cost-effectiveness of web development [KRR14].

Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15a], Big Data, Apps, and Service Ecosystems bring attention to aspects like responsiveness, privacy, and open platforms. Regardless of the application domain, developers of such systems need robust methods and efficient, easy-to-use languages and tools [KRS12].

We tackle these challenges by perusing a model-based, generative approach [PR13]. At the core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale.

We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our development platforms. New services, e.g., for collecting data from temperature sensors, cars, etc. are now easily developed and deployed, e.g., in production or Internet-of-Things environments.

Security aspects and architectures of cloud services for the digital me in a privacy-aware environment are addressed in [ELR+17].

Model-Driven Engineering of Information Systems & the MontiGem Tool

Information Systems provide information to different user groups as the main system goal. Using our experiences in the model-based generation of code with MontiCore [KRV10, HKR21], we developed several generators for such data-centric information systems.

MontiGem [AMN+20a] is a specific generator framework for data-centric business applications that uses standard models from UML/P optionally extended by GUI description models

as sources [GMN+20]. While the standard semantics of these modeling languages remains untouched, the generator produces a lot of additional functionality around these models. The generator is designed flexible, modular, and incremental, handwritten and generated code pieces are well integrated [GHK+15a, NRR15a], tagging of existing models is possible [GLRR15], e.g., for the definition of roles and rights or for testing [DGH+18].

We are using MontiGem for financial management [GHK+20, ANV+18], for creating digital twin cockpits [DMR+20], and various industrial projects. MontiGem makes it easier to create low-code development platforms for digital twins [DHM+22]. When using additional DSLs, we can develop assistive systems providing user support based on goal models [MRV20], privacy-preserving information systems using privacy models and purpose trees [MNRV19], and process-aware digital twin cockpits using BPMN models [BMR+22].

We have also developed an architecture of cloud services for the digital me in a privacy-aware environment [ELR+17] and a method for retrofitting generative aspects into existing applications [DGM+21].

[ABH+16] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Model-Driven Separation of Concerns for Service Robotics. In *International Workshop on Domain-Specific Modeling (DSM'16)*, pages 22–27. ACM, October 2016.

[ABH+17] Kai Adam, Arvid Butting, Robert Heim, Oliver Kautz, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. *Modeling Robotics Tasks for Better Separation of Concerns, Platform-Independence, and Reuse.* Aachener Informatik-Berichte, Software Engineering, Band 28. Shaker Verlag, December 2017.

[ABK+17] Kai Adam, Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Executing Robot Task Models in Dynamic Environments. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[AHRW17] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Engineering Robotics Software Architectures with Exchangeable Model Transformations. In *International Conference on Robotic Computing (IRC'17)*, pages 172–179. IEEE, April 2017.

[AHRW17b] Kai Adam, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Modeling Robotics Software Architectures with Modular Model Transformations. *Journal of Software Engineering for Robotics (JOSER)*, 8(1):3–16, 2017.

[AKK+21] Abdallah Atouani, Jörg Christian Kirchhof, Evgeny Kusmenko, and Bernhard Rumpe. Artifact and Reference Models for Generative Machine Learning Frameworks and Build Systems. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 55–68. ACM, October 2021.

[AMN+20a] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In *40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19)*, LNI P-304, pages 59–66. Gesellschaft für Informatik e.V., May 2020.

[ANV+18] Kai Adam, Lukas Netz, Simon Varga, Judith Michael, Bernhard Rumpe, Patricia Heuser, and Peter Letmathe. Model-Based Generation of Enterprise Information Systems. In Michael Fellmann and Kurt Sandkuhl, editors, *Enterprise Modeling and Information Systems Architectures (EMISA'18)*, CEUR Workshop Proceedings 2097, pages 75–79. CEUR-WS.org, May 2018.

[BBC+18] Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Carlos Fernandez Musoles, Padraig Gleeson, Dan F. M. Goodman, Michael Hines, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, and Jochen Martin Eppler. Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Journal Frontiers in Neuroinformatics*, 12, 2018.

[BBD+21b] Fabian Becker, Pascal Bibow, Manuela Dalibor, Aymen Gannouni, Viviane Hahn, Christian Hopmann, Matthias Jarke, Istvan Koren, Moritz Kröger, Johannes Lipp, Judith Maibaum, Judith Michael, Bernhard Rumpe, Patrick Sapel, Niklas Schäfer, Georg J. Schmitz, Günther Schuh, and Andreas Wortmann. A Conceptual Model for Digital Shadows in Industry and its Application. In Aditya Ghose, Jennifer Horkoff, Vitor E. Silva Souza, Jeffrey Parsons, and Joerg Evermann, editors, *Conceptual Modeling, ER 2021*, pages 271–281. Springer, October 2021.

[BBD+21a] Tim Bolender, Gereon Bürvenich, Manuela Dalibor, Bernhard Rumpe, and Andreas Wortmann. Self-Adaptive Manufacturing with Digital Twins. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 156–166. IEEE Computer Society, May 2021.

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BBR20] Manfred Broy, Wolfgang Böhm, and Bernhard Rumpe. Advanced Systems Engineering - Die Systeme der Zukunft. White paper, fortiss. Forschungsinstitut für softwareintensive Systeme, Munich, July 2020.

[BCGR09] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BDH+20] Pascal Bibow, Manuela Dalibor, Christian Hopmann, Ben Mainz, Bernhard Rumpe, David Schmalzing, Mauritius Schmitz, and Andreas Wortmann. Model-Driven Development of a Digital Twin for Injection Molding. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'20)*, Lecture Notes in Computer Science 12127, pages 85–100. Springer International Publishing, June 2020.

[BDJ+22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil van der Aalst, Klaus Wehrle, Andreas

Wortmann, and Martina Ziefle. A Computer Science Perspective on Digital Transformation in Production. *Journal ACM Transactions on Internet of Things*, 3:1–32, February 2022.

[BDL+18] Arvid Butting, Manuela Dalibor, Gerrit Leonhardt, Bernhard Rumpe, and Andreas Wortmann. Deriving Fluent Internal Domain-specific Languages from Grammars. In *International Conference on Software Language Engineering (SLE'18)*, pages 187–199. ACM, 2018.

[BDR+21] Christian Brecher, Manuela Dalibor, Bernhard Rumpe, Katrin Schilling, and Andreas Wortmann. An Ecosystem for Digital Shadows in Manufacturing. In *54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0*. Elsevier, September 2021.

[BEH+20] Arvid Butting, Robert Eikermann, Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. A Library of Literals, Expressions, Types, and Statements for Compositional Language Design. *Journal of Object Technology (JOT)*, 19(3):3:1–16, October 2020.

[BEK+18b] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Modeling Language Variability with Reusable Language Components. In *International Conference on Systems and Software Product Line (SPLC'18)*. ACM, September 2018.

[BEK+19] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Composition of Independent Language Features. *Journal of Systems and Software (JSS)*, 152:50–69, June 2019.

[Ber10] Christian Berger. *Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles*. Aachener Informatik-Berichte, Software Engineering, Band 6. Shaker Verlag, 2010.

[BGH+97] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Towards a Precise Semantics for Object-Oriented Modeling Techniques. In Jan Bosch and Stuart Mitchell, editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader*, LNCS 1357. Springer Verlag, 1997.

[BGH+98a] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. *Journal Computer Standards & Interfaces*, 19(7):335–345, November 1998.

[BGH+98b] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BGRW17] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. Taming the Complexity of Model-Driven Systems Engineering Projects. In *Part of the Grand Challenges in Modeling (GRAND'17) Workshop*, July 2017.

[BGRW18] Arvid Butting, Timo Greifenberg, Bernhard Rumpe, and Andreas Wortmann. On the Need for Artifact Models in Model-Driven Systems Engineering Projects. In Martina Seidl and Steffen Zschaler, editors, *Software Technologies: Applications and Foundations*, LNCS 10748, pages 146–153. Springer, January 2018.

[BHH+17] Arvid Butting, Arne Haber, Lars Hermerschmidt, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 53–70. Springer, July 2017.

[BHK+17] Arvid Butting, Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Classification of Dynamic Reconfiguration in Component and Connector Architecture Description Languages. In *Proceedings of MODELS 2017. Workshop ModComp*, CEUR 2019, September 2017.

[BHK+21] Tobias Brockhoff, Malte Heithoff, István Koren, Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, Merih Seran Uysal, Wil M. P. van der Aalst, and Andreas Wortmann. Process Prediction with Digital Twins. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 182–187. ACM/IEEE, October 2021.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BHR+18] Arvid Butting, Steffen Hillemacher, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. Shepherding Model Evolution in Model-Driven Development. In *Joint Proceedings of the Workshops at Modellierung 2018 (MOD-WS 2018)*, CEUR Workshop Proceedings 2060, pages 67–77. CEUR-WS.org, February 2018.

[BHR+21] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Compositional Modelling Languages with Analytics and Construction Infrastructures Based on Object-Oriented Techniques - The MontiCore Approach. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 217–234. Springer, July 2021.

[BJRW18] Arvid Butting, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Translating Grammars to Accurate Metamodels. In *International Conference on Software Language Engineering (SLE'18)*, pages 174–186. ACM, 2018.

[BKL+18] Christian Brecher, Evgeny Kusmenko, Achim Lindt, Bernhard Rumpe, Simon Storms, Stephan Wein, Michael von Wenckstern, and Andreas Wortmann. Multi-Level Modeling Framework for Machine as a Service Applications Based on Product Process Resource Models. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control (ISCSIC'18)*. ACM, September 2018.

[BKR+20] Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla. *Towards an Isabelle Theory for distributed, interactive systems - the untimed case.* Aachener Informatik Berichte, Software Engineering, Band 45. Shaker Verlag, March 2020.

[BKRW17a] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Architectural Programming with MontiArcAutomaton. In *12th International Conference on Software Engineering Advances (ICSEA 2017)*, pages 213–218. IARIA XPS Press, May 2017.

[BKRW17] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Semantic Differencing for Message-Driven Component & Connector Architectures. In *International Conference on Software Architecture (ICSA'17)*, pages 145–154. IEEE, April 2017.

[BKRW19] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Continuously Analyzing Finite, Message-Driven, Time-Synchronous Component & Connector Systems During Architecture Evolution. *Journal of Systems and Software (JSS)*, 149:437–461, March 2019.

[BMR+22] Dorina Bano, Judith Michael, Bernhard Rumpe, Simon Varga, and Matthias Weske. Process-Aware Digital Twin Cockpit Synthesis from Event Logs. *Journal of Computer Languages (COLA)*, 70, June 2022.

[BPR+20] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. A Compositional Framework for Systematic Modeling Language Reuse. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 35–46. ACM, October 2020.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12b] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[BRS+15] Arvid Butting, Bernhard Rumpe, Christoph Schulze, Ulrike Thomas, and Andreas Wortmann. Modeling Reusable, Platform-Independent Robot Assembly Processes. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob 2015)*, 2015.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15a] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty H.C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Aßmann, editors, *Models@run.time*, LNCS 8378, pages 101–136. Springer International Publishing, Switzerland, 2014.

[CFJ+16] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, November 2016.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[DEKR19] Imke Drave, Robert Eikermann, Oliver Kautz, and Bernhard Rumpe. Semantic Differencing of Statecharts for Object-oriented Systems. In Slimane Hammoudi, Luis Ferreira Pires, and Bran Selić, editors, *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'19)*, pages 274–282. SciTePress, February 2019.

[DGH+18] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Matthias Markthaler, Bernhard Rumpe, and Andreas Wortmann. Model-Based Testing of Software-Based System Functions. In *Conference on Software Engineering and Advanced Applications (SEAA'18)*, pages 146–153, August 2018.

[DGH+19] Imke Drave, Timo Greifenberg, Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Matthias Markthaler, Philipp Orth, Karin Samira Salman, Johannes Richenhagen, Bernhard Rumpe, Christoph Schulze, Michael Wenckstern, and Andreas Wortmann. SMArDT modeling for automotive software testing. *Journal on Software: Practice and Experience*, 49(2):301–328, February 2019.

[DGM+21] Imke Drave, Akradii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. A Methodology for Retrofitting Generative Aspects in Existing Applications. *Journal of Object Technology (JOT)*, 20:1–24, November 2021.

[DHH+20] Imke Drave, Timo Henrich, Katrin Hölldobler, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Modellierung, Verifikation und Synthese von validen Planungszuständen für Fernsehausstrahlungen. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 173–188. Gesellschaft für Informatik e.V., February 2020.

[DHM+22] Manuela Dalibor, Malte Heithoff, Judith Michael, Lukas Netz, Jérôme Pfeiffer, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Generating Customized Low-Code Development Platforms for Digital Twins. *Journal of Computer Languages (COLA)*, 70, June 2022.

[DKMR19] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. Semantic Evolution Analysis of Feature Models. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 245–255. ACM, September 2019.

[DMR+20] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In Gillian Dobbie, Ulrich Frank, Gerti Kappel, Stephen W. Liddle, and Heinrich C. Mayr, editors, *Conceptual Modeling*, pages 377–387. Springer International Publishing, October 2020.

[DRW+20] Imke Drave, Bernhard Rumpe, Andreas Wortmann, Joerg Berroth, Gregor Hoepfner, Georg Jacobs, Kathrin Spuetz, Thilo Zerwas, Christian Guist, and Jens Kohl. Modeling Mechanical Functional Architectures in SysML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 79–89. ACM, October 2020.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[EFLR99a] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. The UML as a Formal Modeling Notation. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language. «UML»'98: Beyond the Notation*, LNCS 1618, pages 336–348. Springer, Germany, 1999.

[EJK+19] Rolf Ebert, Jahir Jolianis, Stefan Kriebel, Matthias Markthaler, Benjamin Pruenster, Bernhard Rumpe, and Karin Samira Salman. Applying Product Line Testing for the Electric Drive System. In Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi, editors, *International Systems and Software Product Line Conference (SPLC'19)*, pages 14–24. ACM, September 2019.

[ELR+17] Robert Eikermann, Markus Look, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Architecting Cloud Services for the Digital me in a Privacy-Aware Environment. In Ivan Mistrik, Rami Bahsoon, Nour Ali, Maritta Heisel, and Bruce Maxim, editors, *Software Architecture for Big Data and the Cloud*, chapter 12, pages 207–226. Elsevier Science & Technology, June 2017.

[FEL+98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Journal Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FIK+18] Christian Frohn, Petyo Ilov, Stefan Kriebel, Evgeny Kusmenko, Bernhard Rumpe, and Alexander Ryndin. Distributed Simulation of Cooperatively Interacting Vehicles. In *International Conference on Intelligent Transportation Systems (ITSC'18)*, pages 596–601. IEEE, 2018.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. Der Energie-Navigator - Performance-Controlling für Gebäude und Anlagen. *Technik am Bau (TAB) - Fachzeitschrift für Technische Gebäudeausrüstung*, Seiten 36-41, März 2011.

[FLP+11b] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FND+98] Max Fuchs, Dieter Nazareth, Dirk Daniel, and Bernhard Rumpe. BMW-ROOM An Object-Oriented Method for ASCET. In *SAE'98, Cobo Center (Detroit, Michigan, USA)*, Society of Automotive Engineers, 1998.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference (IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GHK+08a] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, Informatik Bericht 2008-02. TU Braunschweig, 2008.

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85. SciTePress, 2015.

[GHK+15a] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk

Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 112–132. Springer, 2015.

[GHK+20] Arkadii Gerasimov, Patricia Heuser, Holger Ketteniß, Peter Letmathe, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In Judith Michael and Dominik Bork, editors, *Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers*, pages 22–30. CEUR Workshop Proceedings, February 2020.

[GHR17] Timo Greifenberg, Steffen Hillemacher, and Bernhard Rumpe. *Towards a Sustainable Artifact Model: Artifacts in Generator-Based Model-Driven Projects*. Aachener Informatik-Berichte, Software Engineering, Band 30. Shaker Verlag, December 2017.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0: Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKR+17] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[GKR+96] Radu Grosu, Cornel Klein, Bernhard Rumpe, and Manfred Broy. State Transition Diagrams. Technical report, TU Munich, 1996.

[GLPR15] Timo Greifenberg, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Energieeffiziente Städte - Herausforderungen und Lösungen aus Sicht des Software Engineerings. In Linnhoff-Popien, Claudia and Zaddach, Michael and Grahl, Andreas, Editor, *Marktplätze im Umbruch: Digitale Strategien für Services im Mobilen Internet*, Xpert.press, Kapitel 56, Seiten 511-520. Springer Berlin Heidelberg, April 2015.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GMN+20] Arkadii Gerasimov, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In Bonnie Anderson, Jason Thatcher, and Rayman Meservy, editors, *25th Americas Conference on Information Systems (AMCIS 2020)*, AIS Electronic Library (AISeL), pages 1–10. Association for Information Systems (AIS), August 2020.

[GMR+16] Timo Greifenberg, Klaus Müller, Alexander Roth, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Modeling Variability in Template-based Code Generators for Product Line Engineering. In *Modellierung 2016 Conference*, LNI 254, pages 141–156. Bonner Köllen Verlag, March 2016.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[Gre19] Timo Greifenberg. *Artefaktbasierte Analyse modellgetriebener Softwareentwicklungsprojekte*. Aachener Informatik-Berichte, Software Engineering, Band 42. Shaker Verlag, August 2019.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR09] Hans Grönniger, Jan Oliver Ringert, and Bernhard Rumpe. System Model-based Definition of Modeling Language Semantics. In *Proc. of FMOODS/FORTE 2009, LNCS 5522*, Lisbon, Portugal, 2009.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[Her19] Lars Hermerschmidt. *Agile Modellgetriebene Entwicklung von Software Security & Privacy*. Aachener Informatik-Berichte, Software Engineering, Band 41. Shaker Verlag, June 2019.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15a] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Journal Future Generation Computer Systems*, 56:701–718, 2015.

[HHR+15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'15)*, CEUR Workshop Proceedings 1463, pages 18–23, 2015.

[HJK+21] Steffen Hillemacher, Nicolas Jäckel, Christopher Kugler, Philipp Orth, David Schmalzing, and Louis Wachtmeister. Artifact-Based Analysis for the Development of Collaborative Embedded Systems. In *Model-Based Engineering of Collaborative Embedded Systems*, pages 315–331. Springer, January 2021.

[HJRW20] Katrin Hölldobler, Nico Jansen, Bernhard Rumpe, and Andreas Wortmann. Komposition Domänenspezifischer Sprachen unter Nutzung der MontiCore Language Workbench, am Beispiel SysML 2. In Dominik Bork, Dimitris Karagiannis, and Heinrich C. Mayr, editors, *Modellierung 2020*, pages 189–190. Gesellschaft für Informatik e.V., February 2020.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HKR+16] Robert Heim, Oliver Kautz, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In *Software Architecture - 10th European Conference (ECSA'16)*, LNCS 9839, pages 175–182. Springer, December 2016.

[HKR21] Katrin Hölldobler, Oliver Kautz, and Bernhard Rumpe. *MontiCore Language Workbench and Library Handbook: Edition 2021.* Aachener Informatik-Berichte, Software Engineering, Band 48. Shaker Verlag, May 2021.

[HLN+15a] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Composition of Heterogeneous Modeling Languages. In *Model-Driven Engineering and Software Development*, Communications in Computer and Information Science 580, pages 45–66. Springer, 2015.

[HLN+15] Arne Haber, Markus Look, Pedram Mir Seyed Nazari, Antonio Navarro Perez, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of Heterogeneous Modeling Languages via Extensible and Composable Language Components. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 19–31. SciTePress, 2015.

[HMR+19] Katrin Hölldobler, Judith Michael, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Innovations in Model-based Software and Systems Engineering. *Journal of Object Technology (JOT)*, 18(1):1–60, July 2019.

[HNRW16] Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. Compositional Language Engineering using Generated, Extensible, Static Type Safe Visitors. In *Conference on Modelling Foundations and Applications (ECMFA)*, LNCS 9764, pages 67–82. Springer, July 2016.

[Hoe18] Katrin Hölldobler. *MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 36. Shaker Verlag, December 2018.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer Journal*, 37(10):64–72, October 2004.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR10] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Towards Architectural Programming of Embedded Systems. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VI*, Informatik-Bericht 2010-01, pages 13 – 22. fortiss GmbH, Germany, 2010.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181-192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HRW18] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Software Language Engineering in the Large: Towards Composing and Deriving Languages. *Journal Computer Languages, Systems & Structures*, 54:386–405, 2018.

[JPR+22] Nico Jansen, Jerome Pfeiffer, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. The Language of SysML v2 under the Magnifying Glass. *Journal of Object Technology (JOT)*, 21:1–15, July 2022.

[JWCR18] Rodi Jolak, Andreas Wortmann, Michel Chaudron, and Bernhard Rumpe. Does Distance Still Matter? Revisiting Collaborative Distributed Software Design. *IEEE Software Journal*, 35(6):40–47, 2018.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KKR19] Nils Kaminski, Evgeny Kusmenko, and Bernhard Rumpe. Modeling Dynamic Architectures of Self-Adaptive Cooperative Systems. *Journal of Object Technology (JOT)*, 18(2):1–20, July 2019.

[KKR+22] Jörg Christian Kirchhof, Anno Kleiss, Bernhard Rumpe, David Schmalzing, Philipp Schneider, and Andreas Wortmann. Model-driven Self-adaptive Deployment of Internet of Things Applications with Automated Modification Proposals. *Journal ACM Transactions on Internet of Things*, 3:1–30, November 2022.

[KKRZ19] Jörg Christian Kirchhof, Evgeny Kusmenko, Bernhard Rumpe, and Hengwen Zhang. Simulation as a Service for Cooperative Vehicles. In Loli Burgueño, Alexander Pretschner, Sebastian Voss, Michel Chaudron, Jörg Kienzle, Markus Völter, Sébastien Gérard, Mansooreh Zahedi, Erwan Bousse, Arend Rensink, Fiona Polack, Gregor Engels, and Gerti Kappel, editors, *Proceedings of MODELS 2019. Workshop MASE*, pages 28–37. IEEE, September 2019.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KMA+16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, and Bernhard Rumpe. VCU: The Three Dimensions of Reuse. In *Conference on Software Reuse (ICSR'16)*, LNCS 9679, pages 122–137. Springer, June 2016.

[KMP+21] Hendrik Kausch, Judith Michael, Mathias Pfeiffer, Deni Raco, Bernhard Rumpe, and Andreas Schweiger. Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. In *Aerospace Europe Conference 2021 (AEC 2021)*. Council of European Aerospace Societies (CEAS), November 2021.

[KMR+20] Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 90–101. ACM, October 2020.

[KMR21] Jörg Christian Kirchhof, Lukas Malcher, and Bernhard Rumpe. Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In Eli Tilevich and Coen De Roover, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 21)*, pages 197–209. ACM, October 2021.

[KMS+18] Stefan Kriebel, Matthias Markthaler, Karin Samira Salman, Timo Greifenberg, Steffen Hillemacher, Bernhard Rumpe, Christoph Schulze, Andreas Wortmann, Philipp Orth, and Johannes Richenhagen. Improving Model-based Testing in Automotive Software Engineering. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE'18)*, pages 172–180. ACM, June 2018.

[KNP+19] Evgeny Kusmenko, Sebastian Nickels, Svetlana Pavlitskaya, Bernhard Rumpe, and Thomas Timmermanns. Modeling and Training of Neural Processing Systems. In Marouane Kessentini, Tao Yue, Alexander Pretschner, Sebastian Voss, and Loli Burgueño, editors, *Conference on Model Driven Engineering Languages and Systems (MODELS'19)*, pages 283–293. IEEE, September 2019.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[KPRS19] Evgeny Kusmenko, Svetlana Pavlitskaya, Bernhard Rumpe, and Sebastian Stüber. On the Engineering of AI-Powered Systems. In Lisa O'Conner, editor, *ASE19. Software Engineering Intelligence Workshop (SEI19)*, pages 126–133. IEEE, November 2019.

[KR18a] Oliver Kautz and Bernhard Rumpe. On Computing Instructions to Repair Failed Model Refinements. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 289–299. ACM, October 2018.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRR+16] Philipp Kehrbusch, Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, and Christoph Schulze. Interface-based Similarity Analysis of Software Components for the Automotive Industry. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 99–108. ACM, September 2016.

[KRRS19] Stefan Kriebel, Deni Raco, Bernhard Rumpe, and Sebastian Stüber. Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In Stephan Krusche, Kurt Schneider, Marco Kuhrmann, Robert Heinrich, Reiner Jung, Marco Konersmann, Eric Schmieders, Steffen Helke, Ina Schaefer, Andreas Vogelsang, Björn Annighöfer, Andreas Schweiger, Marina Reich, and André van Hoorn, editors, *Proceedings of the Workshops of the Software Engineering Conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19)*, CEUR Workshop Proceedings 2308, pages 87–94. CEUR Workshop Proceedings, February 2019.

[KRRW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113-116. VDI Verlag, 2012.

[KRS+18a] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. In *Conference on Model Driven Engineering Languages and Systems (MODELS'18)*, pages 447 – 457. ACM, October 2018.

[KRS+22] Jörg Christian Kirchhof, Bernhard Rumpe, David Schmalzing, and Andreas Wortmann. MontiThings: Model-driven Development and Deployment of Reliable IoT Applications. *Journal of Systems and Software (JSS)*, 183:1–21, January 2022.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KRW20] Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. Automated semantics-preserving parallel decomposition of finite component and connector architectures. *Automated Software Engineering Journal*, 27:119–151, April 2020.

[Kus21] Evgeny Kusmenko. *Model-Driven Development Methodology and Domain-Specific Languages for the Design of Artificial Intelligence in Cyber-Physical Systems*. Aachener Informatik-Berichte, Software Engineering, Band 49. Shaker Verlag, November 2021.

[LMK+11] Philipp Leusmann, Christian Möllering, Lars Klack, Kai Kasugai, Bernhard Rumpe, and Martina Ziefle. Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In Arkady Zaslavsky, Panos K. Chrysanthis, Dik Lun Lee, Dipanjan Chakraborty, Vana Kalogeraki, Mohamed F. Mokbel, and Chi-Yin Chow, editors, *12th IEEE International Conference on Mobile Data Management (Volume 2)*, pages 61–66. IEEE, June 2011.

[Loo17] Markus Look. *Modellgetriebene, agile Entwicklung und Evolution mehrbenutzerfähiger Enterprise Applikationen mit MontiEE*. Aachener Informatik-Berichte, Software Engineering, Band 27. Shaker Verlag, March 2017.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MKB+19] Felix Mannhardt, Agnes Koschmider, Nathalie Baracaldo, Matthias Weidlich, and Judith Michael. Privacy-Preserving Process Mining: Differential Privacy for Event Logs. *Business & Information Systems Engineering*, 61(5):1–20, October 2019.

[MKM+19] Judith Michael, Agnes Koschmider, Felix Mannhardt, Nathalie Baracaldo, and Bernhard Rumpe. User-Centered and Privacy-Driven Process Mining System Design for IoT. In Cinzia Cappiello and Marcela Ruiz, editors, *Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems*, pages 194–206. Springer, June 2019.

[MM13] Judith Michael and Heinrich C. Mayr. Conceptual modeling for ambient assistance. In *Conceptual Modeling - ER 2013*, LNCS 8217, pages 403–413. Springer, 2013.

[MM15] Judith Michael and Heinrich C. Mayr. Creating a domain specific modelling method for ambient assistance. In *International Conference on Advances in ICT for Emerging Regions (ICTer2015)*, pages 119–124. IEEE, 2015.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer Journal*, 43(5):42–48, May 2010.

[MMR+17] Heinrich C. Mayr, Judith Michael, Suneth Ranasinghe, Vladimir A. Shekhovtsov, and Claudia Steinberger. Model centered architecture. In *Conceptual Modeling Perspectives*, pages 85–104. Springer International Publishing, 2017.

[MNRV19] Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. Towards Privacy-Preserving IoT Systems Using Model Driven Engineering. In Nicolas Ferry, Antonio Cicchetti, Federico Ciccozzi, Arnor Solberg, Manuel Wimmer, and Andreas Wortmann, editors, *Proceedings of MODELS 2019. Workshop MDE4IoT*, pages 595–614. CEUR Workshop Proceedings, September 2019.

[MPRW22] Judith Michael, Jérôme Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. Integration Challenges for Digital Twin Systems-of-Systems. In *10th IEEE/ACM International Workshop on Software Engineering for Systems-of-Systems and Software Ecosystems*, pages 9–12. ACM, May 2022.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CDDiff: Semantic Differencing for Class Diagrams. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming*, pages 230–254. Springer Berlin Heidelberg, 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11f] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR11g] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Summarizing Semantic Model Differences. In Bernhard Schätz, Dirk Deridder, Alfonso Pierantonio, Jonathan Sprinkle, and Dalila Tamzalit, editors, *ME 2011 - Models and Evolution*, October 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views (extended abstract). In Wilhelm Hasselbring and Nils Christian Ehmke, editors, *Software Engineering 2014*, LNI 227, pages 63–64. Gesellschaft für Informatik, Köllen Druck+Verlag GmbH, 2014.

[MRR14b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *International Conference on Software Engineering (ICSE'14)*, pages 95–105. ACM, 2014.

[MRRW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent Extra-Functional Properties Tagging for Component and Connector Models. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16)*, CEUR Workshop Proceedings 1723, pages 19–24, October 2016.

[MRV20] Judith Michael, Bernhard Rumpe, and Simon Varga. Human behavior, goals and model-driven software engineering for assistive systems. In Agnes Koschmider, Judith Michael, and Bernhard Thalheim, editors, *Enterprise Modeling and Information Systems Architectures (EMSIA 2020)*, pages 11–18. CEUR Workshop Proceedings, June 2020.

[MRZ21] Judith Michael, Bernhard Rumpe, and Lukas Tim Zimmermann. Goal Modeling and MDSE for Behavior Assistance. In *Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 370–379. ACM/IEEE, October 2021.

[MS17] Judith Michael and Claudia Steinberger. Context modeling for active assistance. In Cristina Cabanillas, Sergio España, and Siamak Farshidi, editors, *Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017)*, pages 221–234, 2017.

[Naz17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Aachener Informatik-Berichte, Software Engineering, Band 29. Shaker Verlag, June 2017.

[NRR15a] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. Mixed Generative and Handcoded Development of Adaptable Data-centric Business Applications. In *Domain-Specific Modeling Workshop (DSM'15)*, pages 43–44. ACM, 2015.

[NRR16] Pedram Mir Seyed Nazari, Alexander Roth, and Bernhard Rumpe. An Extended Symbol Table Infrastructure to Manage the Composition of Output-Specific Generator Information. In *Modellierung 2016 Conference*, LNI 254, pages 133–140. Bonner Köllen Verlag, March 2016.

[PR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PBI+16] Dimitri Plotnikov, Inga Blundell, Tammo Ippen, Jochen Martin Eppler, Abigail Morrison, and Bernhard Rumpe. NESTML: a modeling language for spiking neurons. In *Modellierung 2016 Conference*, LNI 254, pages 93–108. Bonner Köllen Verlag, March 2016.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[Pin14] Claas Pinkernell. *Energie Navigator: Software-gestützte Optimierung der Energieeffizienz von Gebäuden und technischen Anlagen.* Aachener Informatik-Berichte, Software Engineering, Band 17. Shaker Verlag, 2014.

[Plo18] Dimitri Plotnikov. *NESTML - die domänenspezifische Sprache für den NEST-Simulator neuronaler Netzwerke im Human Brain Project.* Aachener Informatik-Berichte, Software Engineering, Band 33. Shaker Verlag, February 2018.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR97] Jan Philipps and Bernhard Rumpe. Refinement of Information Flow Architectures. In M. Hinchey, editor, *ICFEM'97 Proceedings*, Hiroshima, Japan, 1997. IEEE CS Press.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15.* Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rei16] Dirk Reiß. *Modellgetriebene generative Entwicklung von Web-Informationssystemen.* Aachener Informatik-Berichte, Software Engineering, Band 22. Shaker Verlag, May 2016.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems.* Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, Aachen, Germany, December 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[Rot17] Alexander Roth. *Adaptable Code Generation of Consistent and Customizable Data Centric Applications with MontiDex*. Aachener Informatik-Berichte, Software Engineering, Band 31. Shaker Verlag, December 2017.

[RR11] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *International Journal of Software and Informatics*, 2011.

[RRRW15b] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRS+16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven Semantical Similarity Analysis for Software Product Line Extraction. In *International Systems and Software Product Line Conference (SPLC '16)*, pages 174–183. ACM, September 2016.

[RRSW17] Jan Oliver Ringert, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In *International Conference on Software Engineering: Software Engineering and Education Track (ICSE'17)*, pages 127–136. IEEE, May 2017.

[RRW12] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. A Requirements Modeling Language for the Component Behavior of Cyber Physical Robotics Systems. In Seyff, N. and Koziolek, A., editor, *Modelling and Quality in Requirements Engineering: Essays Dedicated to Martin Glinz on the Occasion of His 60th Birthday*, pages 133–146. Monsenstein und Vannerdat, Münster, 2012.

[RRW13] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13c] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RRW15] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Tailoring the MontiArcAutomaton Component & Connector ADL for Generative Development. In *MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*, pages 41–47. ACM, 2015.

[RSR+99] Bernhard Rumpe, M. Schoenmakers, Ansgar Radermacher, and Andy Schürr. UML + ROOM as a Standard ADL? In Frances M. Titsworth, editor, *Engineering of Complex Computer Systems, ICECCS'99 Proceedings*, IEEE Computer Society, 1999.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04c] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Rum17] Bernhard Rumpe. *Agile Modeling with UML: Code Generation, Testing, Refactoring*. Springer International, May 2017.

[RW18] Bernhard Rumpe and Andreas Wortmann. Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures. In Lohstroh, Marten and Derler, Patricia Sirjani, Marjan, editor, *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, LNCS 10760, pages 383–406. Springer, 2018.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SHH+20] Günther Schuh, Constantin Häfner, Christian Hopmann, Bernhard Rumpe, Matthias Brockmann, Andreas Wortmann, Judith Maibaum, Manuela Dalibor, Pascal Bibow, Patrick Sapel, and Moritz Kröger. Effizientere Produktion mit Digitalen Schatten. *ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb*, 115(special):105–107, April 2020.

[SM18a] Claudia Steinberger and Judith Michael. Towards Cognitive Assisted Living 3.0. In *International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops 2018)*, pages 687–692. IEEE, march 2018.

[SM20] Claudia Steinberger and Judith Michael. Using Semantic Markup to Boost Context Awareness for Assistive Systems. In *Smart Assisted Living: Toward An Open Smart-Home Infrastructure*, Computer Communications and Networks, pages 227–246. Springer International Publishing, 2020.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[TAB+21] Carolyn Talcott, Sofia Ananieva, Kyungmin Bae, Benoit Combemale, Robert Heinrich, Mark Hills, Narges Khakpour, Ralf Reussner, Bernhard Rumpe, Patrizia Scandurra, and Hans Vangheluwe. Composition of Languages, Models, and Analyses. In Heinrich, Robert and Duran, Francisco and Talcott, Carolyn and Zschaler, Steffen, editor, *Composing Model-Based Analysis Tools*, pages 45–70. Springer, July 2021.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Voe11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen.* Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[WCB17] Andreas Wortmann, Benoit Combemale, and Olivier Barais. A Systematic Mapping Study on Modeling for Industry 4.0. In *Conference on Model Driven Engineering Languages and Systems (MODELS'17)*, pages 281–291. IEEE, September 2017.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen.* Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[Wor16] Andreas Wortmann. *An Extensible Component & Connector Architecture Description Infrastructure for Multi-Platform Modeling.* Aachener Informatik-Berichte, Software Engineering, Band 25. Shaker Verlag, November 2016.

[Wor21] Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems.* Aachener Informatik-Berichte, Software Engineering, Band 50. Shaker Verlag, October 2021.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.