# Abstraction and Refinement in Hierarchically Decomposable and Underspecified CPS-Architectures

Bernhard Rumpe and Andreas Wortmann(✉)

Software Engineering, RWTH Aachen University, Aachen, Germany
{rumpe,wortmann}@se-rwth.de

**Abstract.** Model-driven development of cyber-physical systems (CPS) requires modeling techniques based on a well-founded theory that supports addressing development techniques, such as decomposition, refinement and the different notions of time required by its components. Based on an elaborated theory for the modeling of underspecification with respect to nondeterminism, hierarchical composition, refinement that is compatible with composition, and finally proven correct evolution patterns, we discuss how such a theory can be practically applied for the development of CPS. Through an orchestrated efficient simulation, we can identify potential bottlenecks, function failures, hardware risks, *etc.* early. All models as well as the simulation take advantage of the compositionality and the timing refinement properties of the theory. In summary, we discuss how the elaborated theory shapes the simulation and the results.

## 1 Motivation

Rigorous model-driven development requires a well-defined set of integrated modeling notations that allows to define a set of possible implementations as well as a well-founded theory that is able to capture important aspects of the system while at the same time. It should (a) be as *abstract* as possible, (b) allow to specify known properties and to leave unknown properties unspecified, and (c) assist the core techniques in a development process.

A typical development process today has to provide various forms of *underspecification* to allow describing known properties and open issues, to support *refinement* along the development process from abstract requirements to very fine-grained technical specifications, and to *compose* specifications. It is not the composition itself, that is of prime importance, but the ability to *decompose* the problem, solve the smaller problems independently through a chain of refinements, and ultimately compose the solutions. This in particular implies that decomposition and refinement must be compatible. This full compatibility of the composition and refinement is important, because only then a decomposition of the problem leads to component specifications that can be independently


M. Lohstroh et al. (Eds.): Lee Festschrift, LNCS 10760, pp. 383–406, 2018.
https://doi.org/10.1007/978-3-319-95246-8_23

developed and refined. Ultimately, their implementations can be composed being sure that the properties specified originally still hold.

There are not many theories that can serve as the foundation for the development of systems, which potentially consist of a physical and a software part, are inherently distributed, and need to cope with a dynamically changing context, while having to fulfill tasks under given time constraints. Much has been said and written about cyber-physical systems (CPS) [33,34] and how those systems can be described and developed [13,16,30]. Only few theories, such as FOCUS [8] and consideration of superdense time [36,39], can actually serve the challenges discussed above. The more development techniques a theory assists, the more complex it necessarily has to be. Many earlier theories, such as CSP [25], CCS [43], Petri Nets [50], or the $\pi$-calculus [44] yield specific advantages, but unfortunately yield shortcomings in other techniques. Especially the existence of techniques for decomposition and refinement as well as their compatibility are crucial.

In this article, we summarize stream based theory, that emerges from FOCUS [5,8,31,53] and has been elaborated by Manfred Broy and his group over several decades in a larger set of publications. Model-driven development [15,65] can facilitate engineering of CPS [29], but requires implementing the underlying theory properly. Consequently, we also present how the theory is implemented in a tool suite called *MontiArc* [9,18,20], that allows to model various aspects of CPS and to simulate CPS with a focus on the interactions within and to the systems context. We briefly discuss, how this theory and its techniques for time and time refinement are realized and we sketch, how MontiArc models are used for example in robotic applications [24,52]. The key idea of the orchestrated efficient simulation that MontiArc provides, is to early identify potential bottlenecks, function failures, hardware risks, *etc.* All models as well as the simulation take advantage of the compositionality and the timing refinement properties of the theory.

In the following, Sect. 2 summarizes parts of the FOCUS theory, before Sect. 3 presents the MontiArc tool suite with its architecture description language and simulation framework. Afterwards, Sect. 4 discuss the benefits of this approach and Sect. 5 highlights related work. Section 6 concludes.

## 2 Theory of Streams

This section contains a condensed form of stream theory. Literature, such as [7,8,53] gives more detailed motivation and discussion of the properties. Ultimately, employing stream theory as the foundation for model-driven development enables modeling architectures for software-intensive CPS as depicted in Fig. 1 under consideration of time as required by its different components.

### 2.1 Streams and Stream Processing Functions

We consider *components* that only interact through explicit, directed, and typed *communication ports*. Such a component can be atomic or decomposed into

*sub-components*. When composing components, ports are connected through directed *channels*.

A *channel observation* is modeled as *stream* $M^\omega$ of finite or infinitely messages over a message alphabet $M$. Progress of time is modeled by an explicit ✓ (called "*tick*") message assuming that each occurrence of the message denotes the start of the next time slice. Thus $M^\omega_\checkmark$ describes a timed observation of a certain time interval (count the ticks!). A complete observation therefore has to contain infinitely many ticks. Within a time slice any finite sequence of messages including the empty sequence may occur. This models the order of messages, but abstracts away from the concrete time. Time synchronous systems are modeled as $\mathbb{N} \to M$, which is the core embedding in the AutoFocus tool suite [2,3,6,26–28]. If messages are optional, $\mathbb{N} \to M \cup \{\bot\}$ is used and $\bot$ is a pseudo message describing the absence of a real message.

There are various forms of mappings of one timing domain into the other as well as many operations on streams [53]. It is possible to choose the form of streams that fit the modeling interests best, but we mostly use $M^\omega$ in the following. In [53], we also embed dense time [39] and Edward Lee's superdense streams [35,36] into the framework.

Ticks partition time into slices, each with a finite sequence of events. The semantics of integrated behavior thus follows the concept of superdense time [35, 39], which, distinguishes between a discrete "time continuum" (the global FOCUS time) and "untimed causally-related actions" (a behavior model's actions within the time slice of a component).

One stream describes one behavior observation. A specification of allowed behaviors is therefore described by a *set of streams* in $\wp(M^\omega)$. It is a general

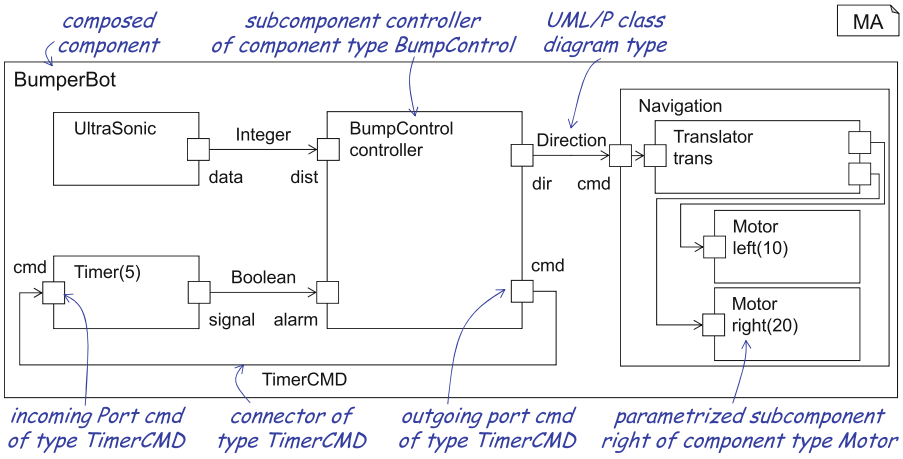


**Fig. 1.** A MontiArc software architecture of a mobile robot. The composed component `BumperBot` contains four sub-components of different types to read sensor data, interpret it, and actuate two motors. The robot explores uncharted territory and avoids obstacles in the process.

principle to use sets as a mechanism for specification and especially *underspecification*, because using a set we can precisely define the allowed properties. Furthermore, *consistency* of a specification corresponds to non-emptiness of a set and *refinement* of a specification corresponds to set inclusion. A set $A$ refines another set $B$ exactly, if $A \subseteq B$. Refinement thus is transitive and reflects that the more information we have the less (mis-)behaviors are possible.

It is not the channel that is of primary interest, but the component and its behavior. The signature of a component is a pair $(I, O)$ of port names from $P$ describing the input and the output. Each port $p \in P$ is typed by the set of messages $M_p$. An observation set of channels $I \subset P$ then is described by a type-preserving mapping of each $i \in I$ to $M_i^\omega$. In short, this mapping is called $\overrightarrow{I}$.

The behavior of a component can then be modeled as a mathematical object of sort $\overrightarrow{I} \rightarrow \overrightarrow{O}$ that maps input behaviors to output behaviors. Please note, that this function completely embeds temporal behavior, because the mapping does not map a single message but has the full observation on its inputs available to determine the full observation on the outputs. However, to be implementable a component may not predict the future, *i.e.,* the output in one time slice may not depend on the input of a later time slice. In the untimed case, *monotonicity* and *continuity*, and in the timed case, *weak* and *strong causality*, are mathematically precise constraints that describe if a function is *realizable*. Fortunately, the forms of streams defined above each form a well-founded CPO (complete partial order) based on prefix ordering that allows defining these constraints.

A realizable function of sort $\overrightarrow{I} \rightarrow \overrightarrow{O}$ describes exactly one possible implementation of the component. We call those *stream processing functions*. Again, we generalize to specifications by using the power set construction, regarding each *component specification* as element of $\wp(\overrightarrow{I} \rightarrow \overrightarrow{O})$. Refinement again is defined as subset.

### 2.2 Composition

Several techniques for modification for components exist, such as *renaming* ports or *hiding* output ports, but of particular interest is *composing* two component specifications, denoted with $\otimes$. Specification composition is defined by pointwise specification of functions and two functions $f, g$ are connected through the channels with same names (and inverted directions). $f \otimes g$ basically is function composition and thus very well understood. Its new output signature is $O = O_f \cup O_g$ and input $I = (I_f \cup I_g) \setminus O$ and thus does not hide connected channels.

Other forms of composition include explicitly named pairs of channels that shall be connected, automatic hiding of connected channels, as well as specialized variants, such as parallel and sequential composition or feedback. All are grounded on the same composition principle. As composition is associative and commutative, it can be generalized to composing any forms of architectures.

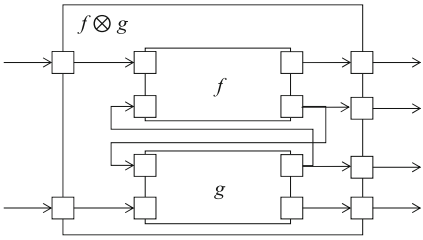


**Fig. 2.** General form of composition.

Composition is well defined in each of the individual streams' domains. And because it is defined pointwise for specifications, properties of the resulting composed specification can be inferred from properties of the individual specifications. For example, the composition is a consistent specification exactly when both components have a consistent specification (Fig. 2).

But most importantly, composition and refinement are compatible, *i.e.,* given three specifications $S, S', T$, where $S'$ is a refinement of $S$, then

$$S' \subseteq S \quad \longrightarrow \quad S' \otimes T \subseteq S \otimes T.$$

Therefore, refinement of any decomposed component leads to a refinement of the overall composition. Refinement means that details on the implementation are decided and more information added and thus less behavior possible. The compatibility of refinement with composition means that once the system is decomposed, each component can be developed and refined independently.

Because decomposition can be applied hierarchically, a complex CPS can be decomposed into individual, atomic, and manageable components.

### 2.3 Description Styles for Components

A mathematical theory such as streams for describing CPS needs to be backed up by more pragmatic styles of denoting specifications. The stream theory does not directly qualify as a specification technique, but serves as a semantic domain [23] for an appropriate set of concrete modeling notations.

Neither infinite streams, nor stream processing functions, nor sets over both should directly be used in mathematical definitions. Instead a structural modeling technique should be available to define the internal decomposition of components. A hierarchy of such structural decompositions finally leads to an *architecture* comprising ports, channels, components and their composition. MontiArc's main modeling sublanguage allows to describe system architectures based on streams. Message types and potentially other, internally used, forms of types must be defined using an appropriate data structure language, *e.g.,* UML class diagrams [17,60]. Behavior of components can be defined in a relational form, using for example the assumption/guarantee style composed of two logic specifications [8], where the assumption restricts the allowed input and the guarantee relates input to output.

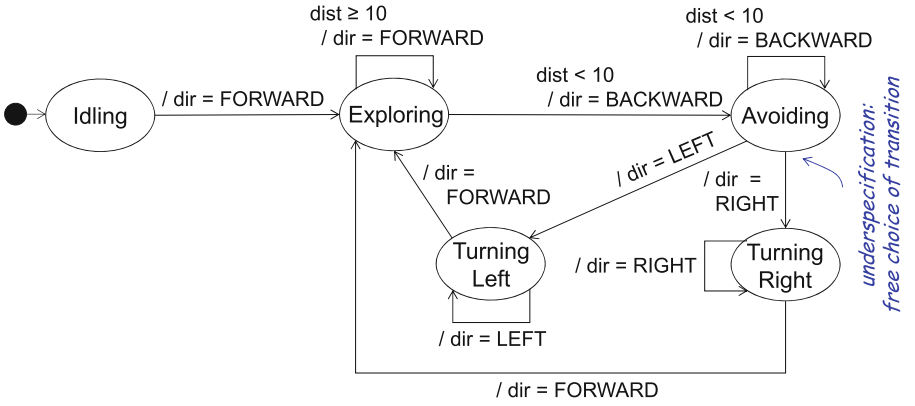


**Fig. 3.** State machine describing the behavior of the `BumpControl` component of the `BumperBot` software architecture (*cf.* Fig. 1).

**State Machines.** Today, state machines are used in various forms, which include Statecharts [21,22], finite or infinite automata, Büchi automata [64], I/O-automata [1,37], or I/O$^\omega$-automata [53,58,59] allow to describe behavior in a stepwise manner, based on an internal state. Dependent on the form of the state machine, different specific properties, such as liveness or completeness, can potentially be described. The important concept of underspecification, which we above realize through power sets, can partially be used within the automaton language directly, using alternate transitions. Indeed is nondeterminism in the state machine specification technique perfectly corresponding to underspecification in the development process and if the developer does not decide, which of the alternatives to be taken, actually the implementation may choose nondeterministically.

Figure 3 depicts a state machine leveraging nondeterminism to specify the behavior of the component `BumpControl` of the `BumperBot` software architecture illustrated in Fig. 1: based on stimuli received through input `dist`, it describes how the systems explores an area until finding an obstacle (states `Exploring` and `Avoiding`). Afterwards it can drive backward, turn left, or turn right (states `Avoiding`, `Turning Left`, and `Turning Right`) until it selects to continue exploring. All decisions following entering state `Avoiding` are based on nondeterministic choice, which is suitable to underspecify CPS properties in different design stages.

I/O$^\omega$ automata are still not a concrete modeling language, but are conceptually rather close. Such an automaton is a tuple $(S, M_{in}, M_{out}, \delta, I)$ with a potentially infinite set of states $S$, input and output alphabet $M_{in}$ and $M_{out}$, a state transition function $\delta \subseteq S \times M_{in} \times S \times M_{out}^{\omega}$ and initial state and output pairs $I \subseteq S \times M_{out}^{\omega}$. An I/O$^\omega$ automaton can easily be mapped to a set of stream processing functions [58].

If the automaton is total, then the component specification is consistent (*i.e.,* a nonempty set of functions). If the automaton is deterministic, then exactly one function is in the semantics. If the automaton is not total, then several choices, such as error completion, full underspecification or ignoring input messages that cannot be handled are available. This all holds for the untimed and timed cases.

Furthermore, there are a larger set of modifications on automata available, such as removing one of several alternate transitions or splitting states, that by application are correct refinements [53,59]. These modifications allow an evolutionary development of atomic component specifications. One of the advantages of state machines is that they can always be directly interpreted as implementation (with more or less choices) and thus used in simulations.

**Architectures.** The composition operation $\otimes$ allows to build hierarchically composed systems. To explicitly understand the architecture, it can be modeled explicitly. A static architecture is minimally modeled by $(A, sub, \sigma, \beta)$, where $A$ denotes the set of components (respectively component identifiers), $sub : A \rightarrow \wp(A)$ the hierarchy of compositions, for $a \in A$, $\sigma(a) = (I_a, O_a)$ is the signature of the component and $\beta(a)$ denotes a behavioral specification of the component in form of a set of stream processing functions $\wp(\overrightarrow{I_a} \rightarrow \overrightarrow{O_a})$. Signatures as well as behavior can now be derived bottom-up through the composition operator as well as specified top-down using for example functional or state based specifications.

It is possible, to use several specification techniques describing different aspects of the same component. Dependent on the form of development process, it may make sense to start with an incomplete assumption/guarantee specification, complete it into a state machine and then hierarchically refine the structure and decompose the overall behavior into a number of interacting components. Semantically, we always know, whether the development steps have been correct, because either they are refinements or we can compare the semantics of a composed architecture and the semantics of a state machine on the basis of the sets of stream processing functions that they define.

### 2.4 Refinement

It is worth to have a deeper look at refinement. *Refinement* is defined as relation between mathematical models that exhibits useful properties. Model $B$ generally is a refinement of $A$, when implementations allowed by $B$ are all correct implementations of $A$. In its simplest form, $A$ and $B$ are sets of implementations themselves and refinement is realized by the subset equation. This holds for stream specifications as well as for specification of components, which are sets of stream processing functions.

The notion of refinement can be extended in two ways: (1) instead of using a pure semantic relation, constructive transformation techniques are defined, and (2) if the signature of the components changes then signature mappings for abstraction and refinement need to be added.

Constructive transformations $T$ can be used instead of using a pure semantic relation $R(.,.)$. They by definition lead to the appropriate refinement. That means for all models $A$ we have $R(A, T(A))$. A sequence of transformations always leads to a refinement of the system. As a consequence, refinement needs to be transitive. The refinement techniques need to be chosen and defined according to the methodical steps that the developer needs. While the refinement relation is defined in a general form concrete transformation steps should be handy, simple and understandable and thus many kinds of small transformation steps are useful.

**Refinement of State Machines Through Transformation.** We demonstrate the general principle of constructive transformations for refinement on the already mentioned refinement concept for state machines as defined in [59]. We repeat the list of concrete refinement steps from [53] in Table 1.

**Table 1.** Refinement transformations preserving or refining semantics of automata models $A = (S, M_{in}, M_{out}, \delta, Init)$ to $T(A) = (S', M'_{in}, M'_{out}, \delta', Init')$

| Transformation | Condition and Description |
|---|---|
| $Init' \subseteq Init$ | Removing initial non-determinism |
| $\delta' \subseteq \delta$ | Removing non-deterministic transitions (with same input in same state); constraint: only for reduction of nondeterminism |
| $\delta' \supseteq \delta$ | Add transitions: removing partiality of accepted input; constraint: not allowed to introduce alternatives to existing transitions |
| $S' \subseteq S$, $\delta' \subseteq \delta$ | Removing states not reachable with any finite or infinite transition sequence |
| $S' \supseteq S$ | Adding states |
| $S \rightsquigarrow S'$ | $S$ replaced by $S'$ with a total, surjective relation that respects $\delta'$ from $S$ to $S'$ (adapting $\delta'$ and $Init'$) |
| $Init \rightsquigarrow Init'$ | Changing initial state where initial output is infinite |
| $\delta \rightsquigarrow \delta'$ | Changing destination state where output is infinite |
| $M_{in} \subseteq M'_{in}$ | Extending input alphabet: semantics preserved for inputs of $M_{in}^{\omega}$ that do not contain any of the new messages |
| $S' = S_{\perp}$, $\delta \subseteq \delta'$ | Chaos complete: adding error state $\perp$, making transition relation total using target state $\perp$, and allowing any output |
| $\delta \supseteq \delta'$ | Compactify: transforming transitions with infinite output to self-loops |

In each case $T(A)$ is a refinement of the original state machine $A$, if we ensure that the context conditions (*i.e.,* well-formedness rules and the application rules for the transformation) are met. Refinement here means, that the semantics

$[\![A]\!]$ and $[\![T(A)]\!]$, which of both sets of stream processing functions, are in the appropriate relation: $[\![A]\!] \supseteq [\![T(A)]\!]$.

As discussed, $(S, M_{in}, M_{out}, \delta, Init)$ is still not a concrete modeling syntax, but it exhibits many more concepts of a concrete modeling language. It will therefore be easier to map a concrete state machine modeling language to these concepts and then understand, what the appropriate evolution steps on state machines are to ensure refinement.

**Refinement of Architectures.** There also is an evolutionary calculus available that allows to modify the given structure of a decomposed component in a controlled way, such that the overall behavior defined by the outside specification is not altered or only refined, when modifying the component internally [48,49]. We call this *glass-box refinement*. This contrasts both, the black-box refinement, where only specifications are considered, as well as the decomposition refinement, where a black-box is decomposed into an architecture of communicating components using a composition operator.

A *decomposition refinement* actually is a modification of the architecture $(A, sub, \sigma, \beta)$ in such a way that a so far atomic component $a \in A$ becomes decomposed by a set of new components. Glass-box refinement allows to modify components and their interconnections and thus leads to calculus like the one presented in Table 2, taken from [48,49], where also the context conditions are precisely defined.

In addition, the papers [40,41,51,56] also have explored to use architecture definitions as incomplete views. That means while syntactically equivalent to an architectural definition, the view only depicts certain components, omits uninteresting channels and also boundaries, how these components are embedded into an architecture. A view based specification therefore corresponds well to the independent modeling of a feature in a high-level form independent of any technical architecture. And those features can in the development process be merged into a complete architecture allowing, *e.g.,* an efficient form of variant management. Again a variety of refinement techniques are possible on views.

**Refinement of Component Signatures Using Mappings.** If the signature of the discussed components change or the set of messages in a set of streams changes, then the specifications are not directly comparable. This happens at many architectural modifications, *e.g.,* if new inputs or outputs are added or a port is renamed. In this case an *abstraction mapping* and a *representation mapping*—we call them $\alpha$ and $\rho$—are necessary to relate the two specifications respectively their semantics. Details of these mappings differ depending on the form of refinement. Again it is mandatory that signature refinements are transitive, which is achieved through function composition on chains of refinement and abstraction mappings.

As simple refinement for two sets of messages $M$, $N$ is defined using an injective $\rho : M \to N$ and $\alpha(\rho(m)) = m$ for $m \in M$. Then $\rho$ is an encoding of the old messages into a potentially more technical representation and $\alpha$ is

**Table 2.** Refinement transformations preserving or refining semantics of architecture models $S = (A, sub, \sigma, \beta)$ to $T(S) = (A', sub', \sigma', \beta')$

| Transformation | Condition and Description |
|---|---|
| $\beta'(a) \subseteq \beta(a)$ | *Behavioral refinement* of the specification for component $a \in A$, usually under an invariant $\Phi$ that is valid on any system execution that has this architecture |
| $A' = A \cup N$ | *Architectural decomposition* of an atomic component $a \in A$, *i.e.,* $sub(a) = \emptyset$, by a set of new components $N \notin A$, where $sub'(a) = N$, $sub'(N) = \emptyset$ and $sub' = sub$ otherwise |
| $\sigma'_o(a) = \sigma_o(a) \cup \{c\}$ | Adding output channel to a component that has previously been hidden internally *i.e.,* $c \in \sigma_o(sub(a))$ |
| $\sigma'_o(a) = \sigma_o(a) \setminus \{c\}$ | Removing an output channel that is not used by sibling components, nor further exported, *i.e.,* for parent $p$ with $a \in sub(p)$: $c \notin \sigma_i(sub(p))$ and $c \notin \sigma_o(p)$ |
| $\sigma'_i(a) = \sigma_i(a) \cup \{c\}$ | Adding input channel that is now available, but unused |
| $\sigma'_i(a) = \sigma_i(a) \setminus \{c\}$ | Removing an input channel of a component. This is only allowed, when the component does not rely on the input channel under an invariant $\Phi$. This can either be checked syntactically (absence of use of $c$) or needs a proof |
| $A' = A \cup \{a\}$ | Adding a component $a$ is always uncritical. The component may be added at any level of the hierarchy and read all available channels. It's output isn't used (yet) and thus the modification is uncritical. ($sub'$ includes $a$, $\beta'$ extended on $a$ as well) |
| $A' = A \setminus \{a\}$ | Removal of a component $a$ is allowed, when the component has no impact, *i.e.,* doesn't emit any channel – $\sigma_o(a) = \emptyset$ – or it's channels are not used anymore (see removing output channels) |
| $A' = A \setminus \{a\}$ | *Expanding component structure* of $a \in A$, where $sub'(p) = sub(p) \setminus \{a\} \cup sub(a)$, leading to an expansion of the internal structure of $a$ into it's father component $p$ |
| $A' = A \cup \{a\}$ | *Folding a sub-component structure* by introducing new component $a \in A$ and embedding a subset $C \subseteq sub(p)$ in component $a$, for instance, $sub'(p) = sub(p) \cup \{a\} \setminus C$ and $sub(a) = C$ |

the corresponding abstraction. All messages in $N \setminus \rho(M)$ are not needed and should therefore not occur in system executions. However, components may react robustly on those messages, for example by ignoring them.

Components $a$ using $M$ as input on a port $p$ may be refined accordingly. With $\rho^c(a)$ and $\alpha^c(a)$, we denote the specifications resulting from the signature change of component $a$ induced by $\rho$ and $\alpha$. Because specifications are sets of stream processing functions, $\rho^c$ and $\alpha^c$ are mappings between sets, resulting

in $\alpha^c(\rho^c(a)) = a$. The latter equality ensures the faithfulness of the encoding representation.

There are many possible forms to extend encodings. We, for example, can use a surjective, but not necessarily injective abstraction $\alpha$, allowing that many messages in $N$ represent the same abstract message in $M$. Then $\rho$ is a relation, but still $(\alpha \otimes \rho)(m)$.

We could represent an abstract message in $M$ by a sequence of messages in $N$. This can be described by $\rho : M^\omega \to N^\omega$ and again $\alpha(\rho(s)) = s$ for $s \in M^\omega$. Again, the encoding does not discuss, what happens with illegitimate sequences of messages, *i.e.,* $s \notin \rho(M^\omega)$, which gives additional freedom when further refining the resulting specification. However, illegitimate sequences of messages should not even occur in a system execution, because through proper refinement of an architecture, the emitting component obeys the same encodings as the receiving component.

If the encoding covers even several channels, *e.g.,* when mapping an 32-bit integer into 32 separate binary channels, then $\rho$ and $\alpha$ will be applied on sets of channels.

Through these various generalizations and the possibility to build chains of encodings $\rho_1(\rho_2(\ldots))$, we finally are able to map abstractly defined components to concrete components and relate their specifications in form of an *U-simulation* (see [4]). U-simulation uses the idea that the input is mapped down via $\rho$ to a concrete representation and the output is mapped back via $\alpha$: The refinement of component $a$ is therefore $\rho^c(a) = \alpha \circ a \circ \rho$. This technique is useful, when a single component is to be refined and shall be used in the original, unchanged context.

If a complete architecture is to be refined, then it is sufficient to define representation mappings for all channels using $\rho$ and apply the representation mapping to all components in an architecture. However, $\rho$ also needs to have an inverse relation with certain properties, to ensure that an encoding is complete and faithful. [4] calls this *refinement under the representation specification* $\rho$ or *downward simulation*. In that article, upward simulation and $\mathrm{U}^{-1}$-simulation are defined also.

Relatively simple forms of refinement, namely the renaming of a channel or the replacement of a set of messages by an equivalent one are easily subsumed under these forms of interaction refinement. Several of the above discussed glass-box modifications for a given architecture can also be derived by applying abstraction and representation mappings on the architectures.

**Refinement of Time.** Time is a very special concept. It is worth to take a deeper look at the possibilities of modifying specifications, that incorporate time. Above we introduced the tick ✓ to model the progress of time. Precisely, in a stream two consecutive ticks represent the beginning and end of a time slice. All time slices in a stream are of equal length, although we do not necessarily need to know the length explicitly. Furthermore, in all streams on all channels ticks model the same progress of time.

Initially, the tick was introduced mainly to model delay. With the tick it became possible to describe, for example, the *merge* function inductively, which previously was not possible. When real-time functions became more important *e.g.,* in the domain of CPS, the tick was also used to represent equidistant progress of time. Formally, the tick is handled like any other message in a stream, which means that stream processing functions may react on progress of time. In particular, we may model timeouts by counting the ticks, which implements clocks.

Timed streams, therefore, have a very similar power of description compared to the concept of superdense event structures [35,36]. All messages within a time slice are known to consecutively follow each other, but nothing is said about the actual progress of time between them. While in the superdense event structure [36], each event has a precise time stamp, in streams only the time slice (and the relative order of events) are known. If real-time comes into play, but the exact timing is not necessary, it should be possible to define time slices small enough to accommodate timed behavior specifications. This abstraction might be useful in specifications especially for underspecification.

Assuming, that a given specification uses a time slice of size $t$. When refining the specification to be able to more precisely describe expected behavior, we might be interested in *refining time* as well, splitting each time slot into $n$ sub-slots. Formally, such a refinement is defined by an abstraction mapping $\alpha : M_{\checkmark}^{\omega} \rightarrow M_{\checkmark}^{\omega}$ that filters each consecutive $n-1$ ticks, while emitting each $n$-th tick. The representation $\rho$ is therefore a relation allowing many different forms of splits for the time slice, *i.e.,* injecting ticks at different places in a stream. Time refinement can also be chained, allowing a hierarchy of time slices.

For simulation purposes, it is interesting to relax the constraint that all ticks model the same time slices. First, we may use channels, where the observed behavior differs from channel to channel. We may even allow timed and untimed channels within the same architecture, which allows us to model system structures and component behaviors as abstract as desired. Formally, we assume a minimal and potentially very small time slice $t$ that is available in the whole system. Each channel is then accompanied with a natural number $n$ (or $\infty$) describing the size of its time slice as multiple $n * t$. For a simple mathematical description, we may use $\checkmark_n$ to denote ticks on a channel with multiplicity $n$.

A component can then accept a variety of timed channels, allowing to be internally decomposed into sub-components of different (synchronous) clocks as well as introducing specification components the main purpose of describing how timing behavior is handled.

There is a lot more theory available, *e.g.,* there are interesting techniques to refine time in a state based specification, where each transition describes an event (including timing events) or describes a time slice [4,58].

Equipped with the above summarized theory, we are in the following looking at the simulation environment provided by MontiArc and how several of the above described techniques are practically realized.

## 3 Architecture Models in MontiArc

MontiArc [9,18,20] is an extensible component & connector ADL [42] allowing to describe the architecture of hierarchically composed components. MontiArc, furthermore, comprises languages for definition of data types and the behavior or of components. MontiArc's components realize stream processing functions that can implement the above discussed timing paradigms. All MontiArc languages are realized as textual modeling languages with the MontiCore [32] language workbench, which supports MontiArc's language extension mechanisms [9]. MontiArc and its variants have been applied to the software engineering of automotive software [19], cloud systems [45], and robotics applications [52] in industrial [24] and academic contexts [54,55].

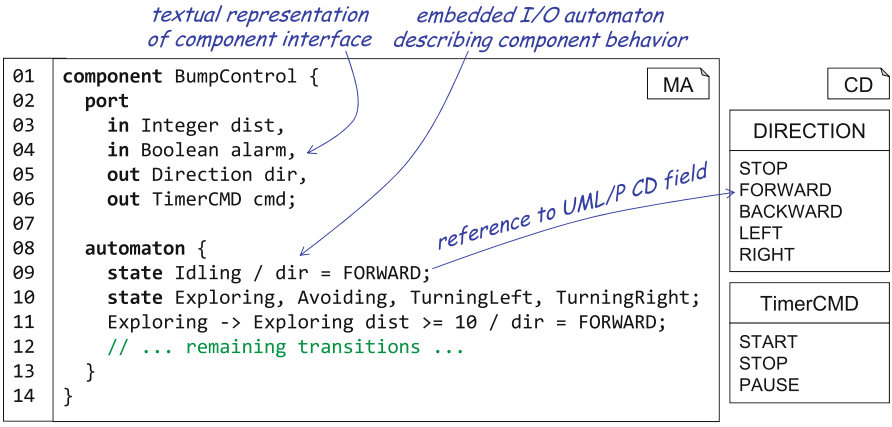


**Fig. 4.** Textual representation of the component `BumpControl` controlling the behavior of the `BumperBot` architecture using an embedded $I/O^{\omega}$ automaton emulating the behavior depicted in Fig. 3.

Components, such as `BumperBot` of Fig. 4 directly correspond to sets of stream processing functions. MontiArc architectures support refinement and composition. The outermost component `BumperBot` defines the system boundary and through instantiation relations and establishment of connectors between its subcomponents defines a software architecture in the sense of $(A, sub, \sigma, \beta)$ (*cf.* Sect. 2.3). With MontiArc, $A$ is the set of components transitively used by the outermost component, $sub$ is characterized by the instantiation relation of the contained components, $\sigma$ is defined by their incoming and outgoing ports, and $\beta$ is defined by the behavior models employed by the instantiated components. To this end, MontiArc components yield interfaces of typed, directed input and output ports through which they receive and emit streams of messages to from and to the environment (ll. 2–6). Components also are either composed or atomic: composed components comprise connectors that realize aforementioned communication channels (*cf.* Sect. 2.1) and through which they define

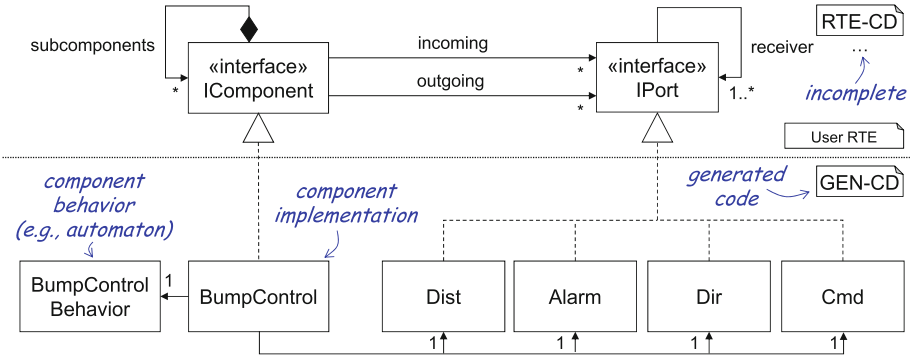


**Fig. 5.** Quintessential interfaces of MontiArc's run-time environment and how they are related to the generated implementation of component `BumpControl` of Fig. 1.

their sub-components' composition (*cf.* Sect. 2.2). Atomic components feature local variables and an $I/O^{\omega}$ automaton describing component behavior (ll. 8–13). An $I/O^{\omega}$ automaton comprises a finite set of states (ll. 9–10), initial variable values, a set of initial states with optional outputs (l. 9), and a set of transitions (ll. 11 ff). Every transition has a source state, a pattern of values read on input ports (inputs) and local variables, a target state, values written to output ports (outputs), and values assigned to local variables (assignments). Inputs, outputs, and assignments may refer to values read from input ports and to values of variables. Embedding other behavior modeling languages is possible [52]. For detailed definitions and well-formedness rules see [20,57].

### 3.1 Transforming MontiArc Models to Executable Java

MontiArc leverages MontiCore's template-based code generation framework [61] to translate component models into executable Java artifacts. To this end, MontiArc parses textual models into abstract syntax trees (ASTs), checks their well-formedness, and applies FreeMarker [63] templates to transform ASTs into Java classes that are compatible to a run-time environment featuring component simulation. This section illustrates this transformation and the next section presents how the Java classes are employed for simulation.

The code generator of MontiArc aims to minimize memory footprint of architectures and operates in the context of a run-time environment (RTE) that provides functionality required by every generated architecture. To this end, it provides various interfaces that generated component code as well as parts of the RTE rely upon. Its quintessential interfaces for describing component structure are `IComponent` and `IPort`, which are implemented by generated component implementations and their ports as depicted in Fig. 5.

Components interact with their environment through sets of incoming and outgoing ports only and can comprise sub-components (composed components only) or behavior implementations (atomic components only) that realize, for

instance, the embedded automata. Each emitting port is connected to a set of receiving ports. This conforms to the FOCUS property that a sender can transmit data to multiple receivers. As sending ports are directly connected to receiving ports, MontiArc does not require to reify connectors (channels) as Java classes. This reduces the number of required objects at runtime and increases scheduling flexibility. Component implementations take care of creating and initializing their sub-components hierarchically according to the corresponding architecture model.

At the core of MontiArc's simulation capabilities is its scheduling infrastructure, which enables simulation of hierarchical architectures of components following different timing paradigms. Each component may carry its own scheduler. Default schedulers are provided, which interact in such a way, that time progress is ensured and all messages are scheduled in their time slot.

Figure 6 depicts its infrastructure but omits the associations already depicted in Fig. 5. Aside from `IComponent` and `IPort`, the schedulers use the following classes and interfaces:

- Interfaces `IOutPort` and `IInPort`: Both interfaces implement `IPort` and enable component developers to send and receive messages respectively.
- Interface `ISimComponent` provides two methods to the scheduler to activate components. Via method `handleMessage(port, message)`, the scheduler invokes processing the passed data `message` on port `port`. The method `handleTick()` to make a component increase its internal clock and emit $\surd$ messages on each outgoing port.
- Abstract class `AComponent` serves a common superclass for generated component classes (such as `BumpControl`) and comprises the component name as well as an error handler.
- Interface `IOutSimPort` provides methods to register receivers (*i.e.,* establish connectors).
- Interface `IInSimPort` enables to setup the containing component and related scheduler to outgoing port instances.
- Additional scheduling-related methods to manipulate the state of ports are provided but omitted in the Figure (*e.g.,* put to sleep, wake up, *etc.*).
- Interface `IScheduler` features the `setupPort(inPort)` method to set up a concrete scheduler and the `registerPort(inPort, msg)` method to trigger scheduling of a certain port and message.
- Interface `IPort` unifies the use of incoming and outgoing ports throughout the generated architecture.
- Interface `IForwardPort` defines incoming ports for decomposed components and forwards messages to the connected incoming ports of the corresponding sub-components.
- Class `Port` is the default port implementation for simulation. To conserve memory, `Port` instances are created for incoming ports of atomic components only. Through `IPort`, instances of the connected incoming ports can be used as outgoing ports and dedicated objects for outgoing port are unnecessary.

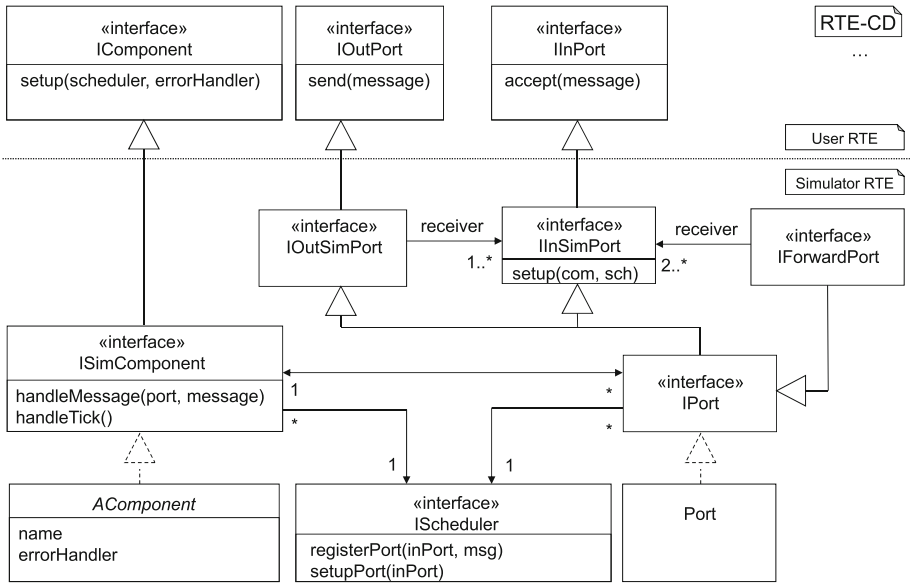


**Fig. 6.** Quintessential classes and interfaces of MontiArc's simulation run-time environment as presented in [18].

Leveraging interfaces to describe MontiArc's scheduling facilitates extending its simulator for different scheduling purposes and timing paradigms. The next section describes how this infrastructure is employed to realize various timing paradigms with MontiArc.

The default scheduling can be individually replaced by specific schedulers, that either know more about the implementation and the order of how messages are processed, or can for example in the simulation be used to experiment with different possible orders to understand how parallel processing respectively interleaving affects the overall outcome. Default scheduling is underspecified in the order of executing the messages (within a given time frame).

The default scheduling is also able to manage cycles of communicating components. Such a cycle needs to be broken up in order to allow progress. In accordance to the theory of Sect. 2, we break each cycle at components that are strongly causal. Strong causality means that the output of a time slice is determined by the inputs of the previous time slice, which means that the component introduces delay, and the calculation of the following components can start already based on the predetermined result of the strongly causal component. If there is a cycle where no component is a strongly causal, the feedback composition will not be well-defined and the simulation would correspondingly get stuck (respectively issues a halting error).

The scheduling order of messages introduces certain form of nondeterminism that may for example occur, if several messages in different channels arrive in the same time slot and are individually processed based on a potentially changed

internal state. Introducing our own schedulers allows to control this form of nondeterminism. Furthermore, for an intensive set of tests of the component interaction, different schedules should be experimented with.

The very same challenge occurs, if the component itself is underspecified, allowing different potential implementations. This is for example the case for nondeterministic state machines, where alternative transitions can be taken with different reactions and different target states. For an extensive simulation, this form of nondeterminism is also to be controlled and scheduled using different alternatives. A typically possible way to control these forms of nondeterminism is to externalize the choice. I.e. instead of nondeterminism, the choice can be controlled by an additional external *oracle*, which may for example be a stream of binary suggestions. I.e. mathematically, we replace and set of stream processing functions $\wp(\overrightarrow{I} \rightarrow \overrightarrow{O})$ by a single deterministic function with an additional input channel $\overrightarrow{\mathbb{B}} \times \overrightarrow{I} \rightarrow \overrightarrow{O})$. The binary decisions $\mathbb{B}$ can be extended to finite or even unbounded choice if necessary. A given architecture can be adapted accordingly, such that each underspecified component and each scheduler receive appropriate oracles. The adapted architecture can be well used for extensive tests in simulations.

### 3.2 Simulating Time in MontiArc Architectures

Simulating time of logically distributed and concurrent components in a single thread requires explicit scheduling, where the schedulers are responsible for message processing and the simulation of time. As discussed above, each component can yield an individual scheduler and a larger variety of scheduling schemes is possible. Each scheduler decides which sub-component executes next and the schedulers synchronize incoming data and ticks received on the incoming ports of components. One strategy is to merge incoming events to a *simulated timed input trace*. This trace is then propagated to scheduled components, which internally process each event and also process timing progress through ✓-events.

As different applications favor different communication timing strategies – embedded applications might favor global clocks whereas cloud systems might benefit from event-driven communication – MontiArc supports all paradigms of time described in Sect. 2. As complex architectures can be composed from components realizing different time paradigms, MontiArc supports registering different schedulers for each composed component. It also provides a default scheduler supporting all three timing paradigms through temporal unification. This is presented in the following.

The foundation of MontiArc simulations are the timed streams discussed in Sect. 2. The timing paradigm of a component determines how the tick messages and data messages of those streams are translated to events, which are propagated to the component implementation. For composed components comprising sub-components of different timing paradigms, the distinct timing behavior is unified to the underlying timed stream paradigm automatically. This entails not forwarding time events to untimed components and forwarding only a single

message per time interval to time-synchronous components. Where special translations between different timing paradigms of sub-components are required, the models have to be adapted to enable proper interaction. This can be achieved by introducing upscaling and downscaling sub-components [8] that translate between different timings in terms of a behavior refinement and serve as adapters between sub-components with different time paradigms.

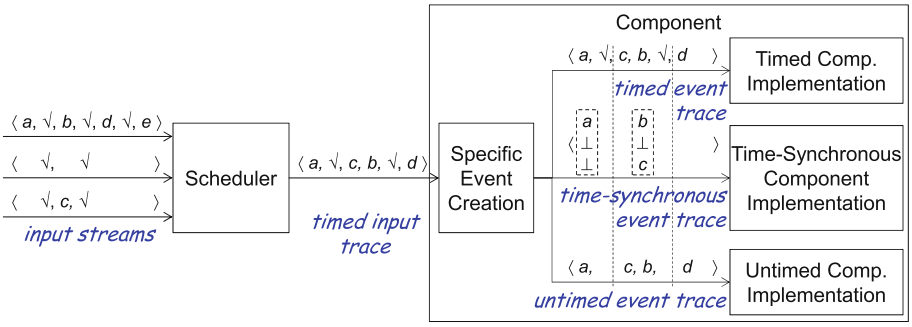


**Fig. 7.** Unification of time-synchronous streams, timed streams, and untimed streams into timed input traces.

MontiArc's default scheduler unifies time in composed components as depicted in Fig. 7: upon receiving a bundle of streams of the component to be scheduled, the scheduler synchronizes these streams into a timed input trace. For each completed time interval in all received input streams, a time event ($\surd$) is present in the produced input trace produced by the scheduler. All data events are propagated to the components input trace in order of their occurrence also. The scheduler then uses the timed input trace to trigger the scheduled component. Time events are raised at the component using its `handleTick()` method, data events are raised using its `handleMessage()` method. The `Specific Event Creation` part of a component then creates timing paradigm specific events that are passed to the concrete implementation of a component.

Unless modeled differently, MontiArc components communicate in a timed fashion. Timed components react to the progress of time as well as data messages on each incoming port. Hence, timed components can produce arbitrary many output messages in a single time slice. Their output is produced in the same time interval in which the triggering input occurred. The `Specific Event Creation` of timed components forwards the received timed input trace from the scheduler to the component implementation. Hence, the timing domain specific event trace directly corresponds to the timed input trace.

Time-synchronous components process up to one input event per time slice and also send up to one message per outgoing port as a reaction to the received input event. Input events of time-synchronous components are tuples holding exactly one message (which may be the empty message $\perp$) per input channel.

The propagation of messages from timed input streams to time-synchronous event traces defines the semantics of time-synchronous components operating over timed streams. The `Specific Event Creation` takes care of creating corresponding tuples to liberate component developers from addressing synchronization.

Untimed components are unaware of timing events, but react to data events only. To this end, the `Specific Event Creation` filters the tick ($\surd$) messages produced by the scheduler as part of the timed input event trace and forwards the result to untimed components accordingly.

Components receiving streams of any timing paradigm can refine the time indicated by these streams as required through decomposing time slices into smaller slices processed by their subcomponents as presented in Sect. 2.4. This corresponds to subcomponents operating in a superdense time where the time continuum can be decomposed until (through architecture decomposition) atomic component perform multiple untimed causally-related actions in a single time slice. Details on realizing the different timing paradigms in MontiArc [18,20] as well as its implementation and tutorials[1] are available.

One of the big advantages of ✓s in the simulation are that the modeled time in the simulation becomes explicit and thus is decoupled from the time necessary to execute the simulation. The simulation can therefore run much larger time frames than it needs to execute the simulation, e.g. necessary for climatic simulations, or vice versa can simulate very tiny timeslots, such as typical for physical atomic processes. Furthermore, when distributing the simulation to many computational cores, then individual cores can run different time frames and can even partially look far into the future, as long as they don't rely on other older parts of the simulation from other cores.

With code generation, hierarchical component instantiation, and extensible scheduling for different timing paradigms in place, MontiArc is suitable to address many challenges arising from engineering software-intensive CPS.

## 4 Discussion

MontiArc supports simulating logically distributed systems of stream processing functions according to different timing paradigms. This enables exploring and validating MontiArc architecture in an agile way. Together with its extensible ADL MontiArc is suitable for rapidly prototyping system models.

The MontiArc simulation realizes the Focus architecture and communication model. Outgoing ports directly transmit messages to connected incoming ports and records of these transmissions correspond to Focus streams that describe the timed communication between sender and receiver. As the MontiArc simulation aims to minimize the memory footprint of architectures at runtime and streams are rarely needed during the execution of a simulation, streams are, by default, not recorded to reduce the amount of allocated memory. However,

[1] See http://www.monticore.de/languages/montiarc/.

for analysis and testing, the relevant ports can be flexibly replaced with test ports that explicitly record transmitted messages in a stream data structure for analysis during or after the simulation execution.

Despite the simulation being executed in a single thread with synchronous blocking method calls, atomic components can be implemented in an event-based fashion. To this end, the MontiArc runtime system prescribes interacting interfaces for components and ports that enable its schedulers to stimulate components with incoming events. Although the message transmission and event propagation by the scheduler require some real time, no simulation time has passed when the control flow returns to a component. Consequently, MontiArc's simulation is logically asynchronous and event-based, which is suitable to a wide range of software-intensive CPS.

## 5 Related Work

A study on architecture description languages discovered over 120 different languages for different kinds of systems operating in different domains [38]. Of these, various languages serve modeling the structure and behavior of software-intensive CPS, including automotive systems [12], avionics [14], consumer electronics [46], and robotics [62]. The languages focus on different aspects and challenges of architecture engineering from academic and industrial perspectives. Overall, architecture description languages rarely are grounded in a well-defined theory. Many prominent ADLs rely on theories realized implicitly through their tooling.

In contrast, AutoFOCUS 3 [26] is a tool suite for developing reactive embedded systems that also bases its semantics on Focus [8]. In contrast to MontiArc, AutoFOCUS 3 cannot leverage the language composition of an underlying language workbench and, hence, does not feature MontiArc's powerful language embedding mechanisms [9]. Further prominent examples of ADLs with well-founded semantics are the $\pi$-ADL [47], LEDA [10], and PiLar [11], all of which rest on the $\pi$-calculus [44], which lacks the powerful properties of Focus regarding composition of refined components.

## 6 Conclusion

We have presented how the Focus theory of stream processing functions can be leveraged to facilitate the model-driven development of cyber-physical systems through early simulation under consideration of different timing paradigms. To this end, we summarized refinement and composition in Focus and showed how automata can employ underspecification to support model-driven specification in early design stages. Based on this theory, we presented the MontiArc architecture modeling tool suite and explained how its code generation and simulation capabilities support engineering software-intensive cyber-physical systems with underspecification and different timing requirements through simulation. We believe that this combination of well-founded theory and practical modeling technique facilitates software engineering of cyber-physical systems.

## References


1. de Alfaro, L., Henzinger, T.A.: Interface automata. SIGSOFT Softw. Eng. Notes **26**(5), 109–120 (2001)
2. Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schtäz, B.: Autofocus 3: tooling concepts for seamless, model-based development of embedded systems. In: 8th International Workshop on Model-based Architecting of Cyber-physical and Embedded Systems, pp. 19–26 (2015)
3. Bauer, A., Romberg, J., Schätz, B.: Integrierte entwicklung von automotive-software mit autofocus. Informatik - Forschung und Entwicklung **19**, 194–205 (2005)
4. Broy, M.: (Inter-)action refinement: the easy way. In: Broy, M. (ed.) Program Design Calculi. NATO ASI F, vol. 118, pp. 121–158. Springer, Heidelberg (1993). https://doi.org/10.1007/978-3-662-02880-3_5
5. Broy, M., Dederich, F., Dendorfer, C., Fuchs, M., Gritzner, T., Weber, R.: The design of distributed systems - an introduction to FOCUS. Technical report, TUM-I9202, SFB-Bericht Nr. 342/2-2/92 A (1993)
6. Broy, M., Huber, F., Schätz, B.: Autofocus - ein werkzeugprototyp zur entwicklung eingebetteter systeme. Informatik-Forschung und Entwicklung **14**(3), 121–134 (1999)
7. Broy, M., Rumpe, B.: Modulare hierarchische modellierung als grundlage der software- und systementwicklung. Informatik-Spektrum **30**(1), 3–18 (2007)
8. Broy, M., Stølen, K.: Specification and Development of Interactive Systems. Focus on Streams, Interfaces, and Refinement. Springer, Heidelberg (2001). https://doi.org/10.1007/978-1-4613-0091-5
9. Butting, A., et al.: Systematic language extension mechanisms for the montiarc architecture description language. In: Anjorin, A., Espinoza, H. (eds.) ECMFA 2017. LNCS, vol. 10376, pp. 53–70. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61482-3_4
10. Canal, C., Pimentel, E., Troya, J.M.: Specification and refinement of dynamic software architectures. In: Donohoe, P. (ed.) Software Architecture. ITIFIP, vol. 12, pp. 107–125. Springer, Boston, MA (1999). https://doi.org/10.1007/978-0-387-35563-4_7
11. Cuesta, C.E., de la Fuente, P., Barrio-Solírzano, M., Beato, M.E.G.: An "abstract process" approach to algebraic dynamic architecture description. J. Log. Algebraic Program. **63**, 177–214 (2005)
12. Debruyne, V., Simonot-Lion, F., Trinquet, Y.: EAST-ADL—an architecture description language. In: Dissaux, P., Filali-Amine, M., Michel, P., Vernadat, F. (eds.) Architecture Description Languages. ITIFIP, vol. 176, pp. 181–195. Springer, Boston, MA (2005). https://doi.org/10.1007/0-387-24590-1_12
13. Derler, P., Lee, E.A., Vincentelli, A.S.: Modeling cyber-physical systems. Proc. IEEE **100**(1), 13–28 (2012)
14. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, Boston (2012)
15. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Future of Software Engineering (FOSE 2007), pp. 37–54, May 2007
16. Giese, H., Rumpe, B., Schätz, B., Sztipanovits, J.: Science and engineering of cyber-physical systems (Dagstuhl seminar 11441). Dagstuhl Rep. **1**(11), 1–22 (2012)

17. OM Group: OMG Unified Modeling Language (OMG UML), Infrastructure version 2.3 (10-05-03) (2010)
18. Haber, A.: MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems. Aachener Informatik-Berichte, Software Engineering, Band, vol. 24. Shaker Verlag, September 2016
19. Haber, A., Rendel, H., Rumpe, B., Schaefer, I.: Evolving delta-oriented software product line architectures. In: Calinescu, R., Garlan, D. (eds.) Monterey Workshop 2012. LNCS, vol. 7539, pp. 183–208. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34059-8_10
20. Haber, A., Ringert, J.O., Rumpe, B.: Montiarc - architectural modeling of interactive distributed and cyber-physical systems. Technical report AIB-2012-03, RWTH Aachen University, February 2012
21. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**, 231–274 (1987)
22. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. NATO ASI F, vol. 13, pp. 477–498. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82453-1_17
23. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of "semantics"? IEEE Comput. **37**(10), 64–72 (2004)
24. Heim, R., Mir Seyed Nazari, P., Ringert, J.O., Rumpe, B., Wortmann, A.: Modeling robot and world interfaces for reusable tasks. In: Intelligent Robots and Systems Conference (IROS 2015), pp. 1793–1798. IEEE (2015)
25. Hoare, C.A.R.: Communicating sequential processes. In: Hansen, P.B. (ed.) The Origin of Concurrent Programming, pp. 413–443. Springer, New York (1978). https://doi.org/10.1007/978-1-4757-3472-0_16
26. Hölzl, F., Feilkas, M.: 13 AutoFocus 3 - a scientific tool prototype for model-based development of component-based, reactive, distributed systems. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) MBEERTS 2007. LNCS, vol. 6100, pp. 317–322. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16277-0_13
27. Huber, F., Schätz, B.: Rapid prototyping with AutoFocus. In: Wolisz, A., Schieferdecker, I., Rennoch, A. (eds.) Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch, pp. 343–352. GMD Verlag, St. Augustin (1997)
28. Huber, F., Schätz, B., Schmidt, A., Spies, K.: AutoFocus—a tool for distributed systems specification. In: Jonsson, B., Parrow, J. (eds.) FTRTFT 1996. LNCS, vol. 1135, pp. 467–470. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61648-9_58
29. Jensen, J.C., Chang, D.H., Lee, E.A.: A model-based design methodology for cyber-physical systems. In: 2011 7th International on Wireless Communications and Mobile Computing Conference (IWCMC), pp. 1666–1671. IEEE (2011)
30. Karsai, G., Sztipanovits, J.: Model-integrated development of cyber-physical systems. Softw. Technol. Embed. Ubiquit. Syst. **5287**, 46–54 (2008)
31. Klein, C., Rumpe, B., Broy, M.: 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, pp. 323–338. Chapmann & Hall (1996)
32. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Int. J. Softw. Tools Technol. Transf. (STTT) **12**(5), 353–372 (2010)

33. Lee, E.A.: Cyber-physical systems-are computing foundations adequate. In: Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap, vol. 2 (2006)
34. Lee, E.A.: Cyber physical systems: design challenges. In: 2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC), pp. 363–369. IEEE (2008)
35. Lee, E.A.: CPS foundations. In: 2010 47th ACM/IEEE Design Automation Conference (DAC), pp. 737–742. IEEE (2010)
36. Lee, E.A.: Constructive models of discrete and continuous physical phenomena. IEEE Access **2**, 1–25 (2014)
37. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Q. **2**, 219–246 (1989)
38. Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., Tang, A.: What industry needs from architectural languages: a survey. IEEE Trans. Softw. Eng. **39**(6), 869–891 (2013)
39. Manna, Z., Pnueli, A.: Verifying hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) HS 1991-1992. LNCS, vol. 736, pp. 4–35. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57318-6_22
40. Maoz, S., Ringert, J.O., Rumpe, B.: Synthesis of component and connector models from crosscutting structural views. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2013), pp. 444–454. ACM, New York (2013)
41. Maoz, S., Ringert, J.O., Rumpe, B.: Verifying component and connector models against crosscutting structural views. In: Software Engineering Conference (ICSE 2014), pp. 95–105. ACM (2014)
42. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. **26**, 70–93 (2000)
43. Milner, R.: Communication and Concurrency, vol. 84. Prentice Hall, Upper Saddle River (1989)
44. Milner, R.: Communicating and Mobile Systems: the $\pi$ Calculus. Cambridge University Press, Cambridge (1999)
45. Navarro Pérez, A., Rumpe, B.: Modeling cloud architectures as interactive systems. In: Model-Driven Engineering for High Performance and Cloud Computing Workshop. CEUR Workshop Proceedings, vol. 1118, pp. 15–24 (2013)
46. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer **33**(3), 78–85 (2000)
47. Oquendo, F.: $\pi$-ADL: an architecture description language based on the higher-order typed $\pi$-calculus for specifying dynamic and mobile software architectures. ACM SIGSOFT Softw. Eng. Notes **29**, 1–14 (2004)
48. Paech, B., Rumpe, B.: State based service description. In: Proceeding of the IFIP TC6 WG6.1 International Workshop on Formal Methods for Open Object-Based Distributed Systems, FMOODS 1997, pp. 293–302. Chapman & Hall Ltd., London (1997)
49. Philipps, J., Rumpe, B.: Refinement of pipe-and-filter architectures. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 96–115. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48119-2_8
50. Reisig, W.: Petri Nets: An Introduction, vol. 4. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-69968-9

51. Ringert, J.O.: Analysis and Synthesis of Interactive Component and Connector Systems. Aachener Informatik-Berichte, Software Engineering, Band, vol. 19. Shaker Verlag (2014)
52. Ringert, J.O., Roth, A., Rumpe, B., Wortmann, A.: Language and code generator composition for model-driven engineering of robotics component & connector systems. J. Softw. Eng. Rob. (JOSER) **6**(1), 33–57 (2015)
53. Ringert, J.O., Rumpe, B.: A little synopsis on streams, stream processing functions, and state-based stream processing. Int. J. Softw. Inform. **5**(1–2), 29–53 (2011)
54. Ringert, J.O., Rumpe, B., Schulze, C., Wortmann, A.: Teaching agile model-driven engineering for cyber-physical systems. In: International Conference on Software Engineering: Software Engineering and Education Track (ICSE 2017), pp. 127–136. IEEE (2017)
55. Ringert, J.O., Rumpe, B., Wortmann, A.: A case study on model-based development of robotic systems using montiarc with embedded automata. In: Giese, H., Huhn, M., Philipps, J., Schätz, B. (eds.) Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme, pp. 30–43 (2013)
56. Ringert, J.O., Rumpe, B., Wortmann, A.: From software architecture structure and behavior modeling to implementations of cyber-physical systems. In: Software Engineering Workshopband (SE 2013). LNI, vol. 215, pp. 155–170 (2013)
57. Ringert, J.O., Rumpe, B., Wortmann, A.: Architecture and behavior modeling of cyber-physical systems with MontiArcAutomaton. No. 20 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag (2014)
58. Rumpe, B.: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Herbert Utz Verlag Wissenschaft, München, Deutschland (1996)
59. Rumpe, B.: Modellierung mit UML. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-642-22413-3
60. Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-33933-7
61. Schindler, M.: Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P. Aachener Informatik-Berichte, Software Engineering, Band, vol. 11. Shaker Verlag (2012)
62. Schlegel, C., Steck, A., Lotz, A.: Model-driven software development in robotics: communication patterns as key for a robotics component model. In: Chugo, D., Yokota, S. (eds.) Introduction to Modern Robotics. iConcept Press (2011)
63. Tedd, L.A., Radjenovic, J., Milosavljevic, B., Surla, D.: Modelling and implementation of catalogue cards using freemarker. Program **43**(1), 62–76 (2009)
64. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, vol. B, pp. 133–191. Elsevier (1990)
65. Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., Czarnecki, K., von Stockfleth, B.: Model-Driven Software Development: Technology, Engineering, Management. Wiley Software Patterns Series. Wiley, Hoboken (2013)