Behavioral Types for Space-aware Systems

Jan Olaf Blech
RMIT University
Melbourne, Australia
{janolaf.blech@rmit.edu.au}

Peter Herrmann
Norwegian University of Science and Technology
Trondheim, Norway
{herrmann@item.ntnu.no}

***Abstract*—Behavioral types for space-aware systems are proposed as a means to facilitate the development, commissioning, maintenance, and refactoring of systems with cyber-physical characteristics. In this paper, we particularly introduce the formal definition of behavioral types that are associated with system components in order to specify their expected behavior. As application domain, we concentrate on systems from industrial automation that encompass recurring behavior.**

I. Introduction

In the industrial automation domain, many systems consist of physically distributed components that cooperate with each other by carrying out recurring behavioral patterns. A typical example is a state-of-the-art assembly line consisting of a series of robots that build complex articles like cars. To work correctly, the behaviors of the components need to fulfill various software and physical behavioral aspects that can be quite diverse and may comprise, for instance, communication protocols, heat emission or spatial occupation (e.g., a robot adding a part to a car must perform trajectories such that the car's carriage is not damaged).

To handle the complexity and diversity of specifying component and system behaviors, we introduce *space-aware behavioral types* that allow us to capture both software and physical aspects. As with types in traditional programming languages, e.g., primitive datatypes and their composition, the behavioral types can be used to eliminate error sources already at the development time of software systems. This is analog to classical static type checks performed by a compiler. Furthermore, we can use the behavioral types to eliminate runtime errors. This resembles dynamic type checks that, in many programming languages, are performed when accessing pointers that reference data of types which cannot be statically determined. Behavioral types also provide additional information about components which can be used for tool-based operations that allow the discovery of components and the dynamic reconfiguration of systems.

The behavioral types introduced in this paper are applicable on different scales such as to express the interaction of the various parts of a single robot or to specify collaboration aspects between different sites (cf. [8]). The limitation to recurring behavior makes it possible to verify behavior by checking only a finite number of situations which eases the use of highly automatic verification tools. Our approach makes it possible to check the following features of a type system:

- *Type compatibility checking* — as known from types of imperative programming languages, e.g., checking whether we can add an integer to a float — with space-aware behavioral types associated to components.
- *Subtyping* allows the replacement of a component with a certain behavioral type t by another component that has a subtype t' of type t. We base subtyping on spatial geometric refinement that can be checked automatically.
- *Type composition* is necessary to infer types of components that are composed of existing components with known types.

In addition, we want to ensure

- *Type conformance*, i.e., the question whether a component really behaves according to its specification: the geometric spatial behavioral type.

A. Motivating Examples

Loading robot: Figure 1 shows two pictures of a robot interacting with a moving device. The robot and the device have spatial behaviors, i.e., their positions in space change during time. At various points on the time scale, that we call *timepoints*, they physically occupy certain spaces that can be characterized by coordinates in a geometric coordinate system. On the one hand, we like to ensure using space-aware behavioral types that the robot does not collide with the moving device. On the other one, we also want to guarantee that the robot grip is coming very close to the device in order to avoid that articles are damaged while being loaded onto the device.

The robot consists of three segments and a tool that are attached to each other via joints. Each of the four robot components has an individual spatial behavior relative to the parts it is attached to. As depicted on the left side of Fig. 1, this spatial behavior can be expressed with a space-aware behavioral type that encodes the movement of a robot part over time. Typically, the behavioral description of each type is relative to a distinct point in the coordinate system. For example, multiple instances of the tool may have the same type, but may be deployed independently in different locations (e.g., segments 1 and 3). Likewise, we can use a behavioral type expressing the behavior of the moving device.

The right side of Fig. 1 shows the composition of the types from the robot's components into a single type representing the behavior of the overall robot. The composed type for the robot takes the relative spatial movements of the segment and

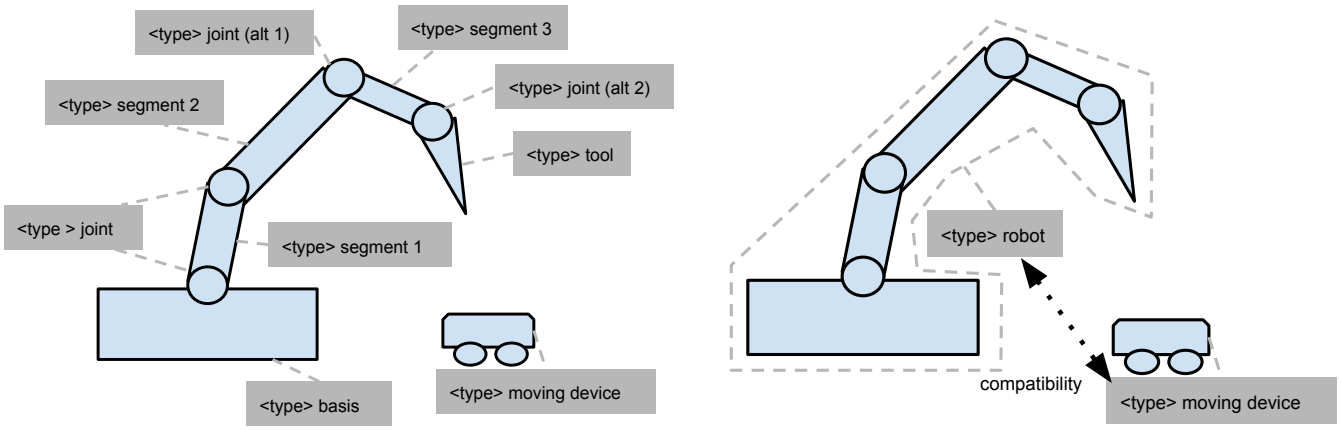

Fig. 1. Behavioral types for a robot

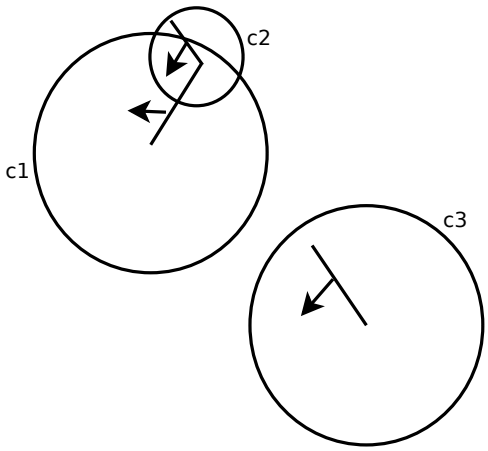

Fig. 2. Spatial behavior of rotating robot arms

tools to each other into account. To verify that the robot does not collide with the moving devices but that its grip comes sufficiently close, we can apply the composed type of the robot instead of the four simple ones referring to its parts. The spatial verifications are carried out by type checking of the composed robot type and the one of the moving device. Using the subtyping feature, we can even replace robot segments by other ones without needing to repeat the type checking proofs of safety properties as long as the replaced segments are in certain relations with the original ones.

Rotating robot arm: Another example of a robot composition is depicted in Figure 2. Here, three components are shown: *c1* is a robot arm. It performs a circular movement around a center point and features a reference point at the outer end that turns counterclockwise. This behavior is captured using a space-aware behavioral type. Another component *c2* also carries out a counterclockwise circular movement albeit with a smaller radius. This is also encoded in a space-aware behavioral type. *c2* gets attached to *c1* via the reference point. By type composition, we can create a behavioral type modeling the joint behavior of *c1* and *c2*.

c3 is also a robot arm, possibly of the same kind as *c1*, that performs the same rotational movement around a different center point. In consequence, the behavioral type of *c3* may be the same as the one of *c1* which, however, refers to another center point.

A typical type checking problem is the decision whether the system composed of *c1* and *c2* can collide with *c3*. For type checking, we compute the least common multiple of the cycle times for each of the three components and compare for each time point whether a collision may occur. The use of time points instead of time intervals requires that the spatial behavior at each time point is a safe approximation of the behavior during the adjacent time intervals. We will discuss this later in detail.

B. Related Work

The idea to use well defined specifications that define the interfaces of software component systems, has been made popular by the design-by-contract approach for software components [31]. More recent work comprises specification and contract languages for component-based systems that have been studied in the context of web services. Process algebra-like languages and associated techniques are studied in [11], [16]. Another algebraic approach to service composition is presented in [18]. In [27], so-called External State Machines (ESMs) are used to specify the interface behavior of functional software building blocks. The ESMs do not only facilitate the integration of the building blocks into their environment but make also compositional model checking of the building blocks possible.

Behavioral types have been studied as interface automata [1] for software components and in the Ptolemy II project [30] for the software part of real-time systems. Further, their use as means for behavioral checks at runtime for component-based systems was investigated in [2].

We proposed a behavioral type system in [9]. In [6], ensuring behavioral type correctness at runtime using techniques from runtime verification was discussed in the context of

Java/OSGi-based applications. Moreover, we studied compatibility checking in [7]. This paper also features a solution for behavioral type coercion for a highly restricted class of behavioral types. Furthermore, we have applied a behavioral types concept to the software part of automation control systems [37] which can be seen as a precursor of the work presented here. Providing a format for spatial behavioral types and means to reason about it is a new contribution of this paper.

Specification of spatial properties has been studied using process algebra-like formalisms [13], [14]. A type system based on this formalism was introduced in [12] for concurrency and resource control. The author presents typing rules and automatic type checking which is not a focus here. Moreover, a verification tool has been developed to check properties based on this formalism in [15]. In contrast, we are interested in developing a solution that fits for industrial robots and related machinery. Therefore, we restrict ourselves to the checking of recurrent behavior in geometric space and concentrate us on tailoring a formalism and compliant checking techniques for this particular domain. Contracts between components with a cyber-physical flavour have been studied in the SPEEDS project [3], [4], [20]. Here, the contracts also take behavior in the form of a transition system into account. In [32] contracts for avionic components are studied.

Reasoning about spatial and geometric constraints is described in, e.g., [5], [25]. A particularly important application domain is robot path planning which has been studied for decades (e.g., [26], [29]). Spatial types are also used for databases, e.g., to manage geometric objects [21] or in Geographic Information Systems [36]. A challenge of these approaches is to guarantee that a reasonable subset of the spatial logic is decidable and, of course, that realistic system models can be checked in an acceptable period of time (e.g., see decidability results in [17]). Logic approaches for hybrid systems (e.g., [19], [34], [35]) provide comprehensive languages and tools for describing cyber-physical systems. In contrast to these works, our focus is stronger aligned with the industrial automation domain and the use as a behavioral type system. The time and geometry focus on the reasoning side of our framework can be complemented by a topological view. This has advantages in areas such as security analysis [33].

As we will show below, the approach presented here fits well to the existing verification technique BeSpaceD [10] that already proved that it can be used to check spatial properties of various systems (see [22], [23], [24]).

C. Overview

Section II introduces our space-aware behavioral types. The underlying semantics and related behavioral types features are discussed in Sect. III. An evaluation is featured in Sect. IV followed by a conclusion in Sect. V.

II. Space-aware Behavioral Types

In general, we describe spatiotemporal behavior for the industrial automation domain by defining which properties hold at which timepoint. Due to the recurrent nature of the behavior, we have to observe only a finite number of timepoints. In Sect. II-A we describe the basic formalism of our behavioral type definitions and introduce certain templates facilitating the use of our method. Thereafter, we discuss the constructors and composition operators in Sect. II-B. In the remainder of this section we justify our modeling choices.

A. Behavioral Descriptions

We use simple logic-based descriptions to define abstract datatypes. These *behavioral descriptions* can be composed of the following operators and predicates:

- *Logical operators:* $\wedge$, $\vee$, and $\neg$ as well as abbreviations such as $\longrightarrow$ and $\bigwedge_{i \in I}$.
- *Predicates that characterize timepoints.* This includes expressions such as timepoints modulo a cycle time — after which the behavior is repeated — and time intervals.
- *Predicates characterizing events.* In addition to the space-aware aspects one can also use events to specify software interaction protocols [9].
- *Predicates indicating nodes and edges in a graph structure.*
- *Predicates indicating occupation of geometric space.*
- *Parameters defining the ownership of space occupation.* Here, spatial occupation behavior is associated with a certain component that *owns* the occupied space.

Our way to associate space occupation with ownership allows us to specify various spatial properties of a component in separation. As already mentioned, examples for such properties that may all refer to the same physical component C, may be C's physical occupation of space, the distribution of heat emitted by C, and the range over which C may broadcast wireless communication messages. These properties can be modeled by separate predicates that all use C as their owner. In consequence, the individual properties can be separately verified by type checking which is carried out based on two different approximation approaches:

- *Overapproximation* means to consider a geometric space that is at least as large as the one that is factually covered by an owner. This fits to properties like the physical occupation of space or the distribution of heat.
- *Underapproximation* refers to a geometric space that is at most as large as the one factually covered. We can use it, for example, to check broadcasting ranges.

The two approaches are closer described in Sect. III.

Templates: Behavioral descriptions encoding a component of the industrial automation / robot domain can follow the templates shown in Fig. 3. The specification features a conjunction over implications. Each implication refers to certain conditions that hold at a certain timepoint and in the presence of events. The conditions can be, for instance, aspects referring to the spatial occupation of a geometric object. Each aspect itself is constructed as predicates of the behavioral description language introduced above. It primarily features constraints on space such as conjunctions of predicates that

$$
\begin{aligned}
&t = 1 \wedge (\neg) \textbf{ event } E_0 \wedge ... \wedge (\neg) \text{ event } E_n \longrightarrow \\
&\qquad \text{Space Occupation Aspect } 1 \wedge ... \wedge \text{Space Occupation Aspect } m \\
&\qquad\qquad \wedge \\
&\qquad\qquad ... \\
&\qquad\qquad \wedge \\
&t = 1 \wedge (\neg) \text{ event } E_0 \wedge ... \wedge (\neg) \textbf{ event } E_n \longrightarrow \\
&\qquad \text{Space Occupation Aspect } h \wedge ... \wedge \text{Space Occupation Aspect } j \\
&\qquad\qquad \wedge \\
&\qquad\qquad ... \\
&\qquad\qquad \wedge \\
&t = cycletime \wedge (\neg) \textbf{ event } E_0 \wedge ... \wedge (\neg) \text{ event } E_n \longrightarrow \\
&\qquad \text{Space Occupation Aspect } 1 \wedge ... \wedge \text{Space Occupation Aspect } m \\
&\qquad\qquad \wedge \\
&\qquad\qquad ... \\
&\qquad\qquad \wedge \\
&t = cycletime \wedge (\neg) \text{ event } E_0 \wedge ... \wedge (\neg) \textbf{ event } E_n \longrightarrow \\
&\qquad \text{Space Occupation Aspect } h \wedge ... \wedge \text{Space Occupation Aspect } j
\end{aligned}
$$

Fig. 3. Template for a behavioral description

indicate the occupation of space for a geometric object. A space occupation aspect is either classified as an over- or an underapproximation.

The template specifies spatial behavior up to the timepoint referring to finishing a recurrent behavior cycle. After the cycle time, the behavioral description is repeated. This, however, does not necessarily always result in the same behavior, since events may be different. Having a cycle time is a typical feature in industrial automation and a key characteristic of Programmable Logic Controllers (PLC) used to control automation facilities (e.g., the IEC 61131-3 and IEC 61499 standards) and for controlling industrial robots.

Behavioral descriptions may be specified by developers manually. However, typical descriptions can comprise several thousand cases. Thus, a preferable way is to specify a system in a simulation or development tool and generate the behavioral description automatically. We have successfully done this using the model-based engineering tool Reactive Blocks [28] as described in [23], [24].

B. Type Constructors and Composition

Type constructors use behavioral descriptions and additional information to create a space-aware behavioral type. We present two kinds of space-aware behavioral types. *Primitive space-aware behavioral types* are often used to capture the behavior of a single atomic component, whereas *composed space-aware behavioral types* are typically applied to capture the behavior of composed systems. However, composed types may also be applied to characterize different aspects of a single atomic component and a primitive type may be used to capture the behavior of a composed system, when no detailed behavior of subcomponents is available or it is not necessary to describe that separately.

Basic space-aware behavioral types: We define three different kinds for the primitive behavioral types:

1) A behavioral description bd may be accompanied by the cycle time ct, after which the behavior is repeated to form a geometric spatial behavioral type using the tuple
$$(bd, ct)$$
2) An extended definition features a geometric offset go which is a point in the geometric space. Likewise, space-aware behavioral types allow to shift the starting time of a cycle by a time offset to in order to allow the reuse of the behavioral description for a component that may be started with a delay. The spatial and starting time impacts of the behavioral description can be described by the following tuples:
$$(bd, go, ct),\ (bd, to, ct) \text{ and } (bd, go, to, ct)$$
3) A component that features a behavior in time and space may be attached to a joint device of another component where this joint device has its own spatiotemporal behavior. This *relative movement* of a component to another is captured in the following type definition: A type may feature a set RP of *reference points* through which other components may be attached to it and a behavioral description is provided with each reference point. For instance, the segments and the tool of the robot introduced in Sect. I-A are attached to each other via reference points. Each reference point exhibits its own spatiotemporal behavior that depends on both, the physical placement of the reference point as well as the behavior of the overall component. In the type constructor, we model the relation between reference points and their behavioral descriptions by the function $\mapsto$ mapping all elements of set RP to the set BD of all possible behavioral descriptions. Thus, if $bd_i \in BD$ is the behavioral description of a reference point $rp_i \in RP$, the formula $\mapsto(rp_i) = bd_i$ holds which

we express as $rp_i \mapsto bd_i$ for convenience. The type constructor is defined as follows:

$$(bd, RP, \mapsto, ct)$$

The behavioral description used in the reference point must only describe the movement of a single point in relation to time and events.

Composed space-aware behavioral types: The behavior of multiple components can be combined, e.g., to form new components or to define alternative types. A way to combine behavior types syntactically is *type composition*. Its semantics is highlighted in the following:

1) The union type $+$ provides an alternative between two different space-aware behavioral types gbt and gbt' each defined as one of the three types introduced above:

$$gbt + gbt'$$

As an example, the intended semantics — a behavioral alternative — of a union of two space-aware behavioral types is given below (lcm denotes the *least common multiple*):

$$(bd, ct) + (bd', ct') \triangleq ((bd \vee bd'), lcm(ct, ct'))$$

2) Compositions as expressed by the operator $\times$ correspond to record types in programming languages:

$$gbt \times gbt'$$

Semantically, that corresponds to the following operation on the behavioral description level:

$$(bd, ct) \times (bd', ct') \triangleq ((bd \wedge bd'), lcm(ct, ct'))$$

Furthermore, as in records, we support an implementation that maps names to behavioral descriptions. This allows us to have record-like field descriptors.

3) Composing structures of components attached to reference points, like in the robot example depicted in Fig. 1, usually leads to lengthy nested behavioral descriptions. To simplify these definitions, we offer non-nested type constructors for such structures. The non-nested variant does not have to be attached to a base component such it does not need to feature a cycle time. The simplified constructor can be used if a structure consisting of composed components is modeled by the basic space-aware behavioral type gbt of kind 3, i.e., $gbt \triangleq (bd, RP, \mapsto, ct)$. We also introduce the set GBT that features the geometric spatial behavior in the remainder of the nested structure, as well as the function $\rightarrow$ mapping the reference points $rp_i \in RP$ in the composed structure to their respective behaviors $gbt_i \in GBT$, i.e., $rp_i \rightarrow gbt_i$. The resulting behavioral type is syntactically defined in the following way:

$$(gbt, RP, \rightarrow)$$

To illustrate this, we regard our motivating example from Sect. I-A and Fig. 1. The composed type for the robot is made up of the behavioral type tt of the tool and the types at_1, at_2, at_3 of the three robot arm segments. The four types can be nested in the following way:

$$\begin{aligned}&(at_1, \{rp_{at_1}\},\\ &\ rp_{at_1} \mapsto (at_2, \{rp_{at_2}\},\\ &\qquad rp_{at_2} \mapsto (at_3, \{rp_{at_3}\}, rp_{at_3} \mapsto (tt))\\ &\qquad)\\ &)\end{aligned}$$

Using our introduced definition, the behavioral type of each segment type at_i with a behavioral description ab_i has the form:

$$(ab_i, \{rp_{at_i}\}, rp_{at_i} \rightarrow rpb_{rp_{at_i}})$$

where rpb_{rp_k} refers to the behavior of a reference point rp_k thereby removing the nested structure.

Our notion of behavioral types takes the intended semantics into account, i.e., the behavior in space and time. Different *syntactic type definitions* which may be grouped into equivalence classes may exist for the same space-aware behavioral type. For instance, by using the symmetry of the union operator in type composition or the symmetry of $\wedge$, we can construct syntactically different type definitions for the same type.

III. Semantics of Space-aware Behavioral Types

To facilitate the verification that objects occupy a certain geometric space in an area, we can use subtyping of the behavioral types of these objects. As described in Sect. II-A, verification of spatial properties can be performed based on both, overapproximation and underapproximation. This is considered by distinguishing subtyping between *overapproximation-refinement aspects* and *underapproximation-refinement aspects*. A space-aware behavioral type T' is a subtype of another type T if and only if the following conditions hold for each spatial aspect and each shared timepoint t:

- For overapproximation-refinement aspects, the space occupation at t specified in T' is geometrically included in T. Thus, overapproximation-oriented spatial proofs (e.g., collision avoidance) that were carried out for a physical component represented by T also hold for a "smaller" one described by T'.
- For underapproximation-refinement aspects, the space occupation at t specified in T is geometrically included in T'. So, underapproximation proofs (e.g., broadcast ranges) done for T hold also for a "larger" T'.
- For both, overapproximation-refinement and underapproximation-refinement aspects hold, that if T comprises unbound reference points, T' incorporates the same unbound reference points, which show an identical behavior.

Subtyping imposes a partial order relation between the space-aware behavioral types since according to our definition the following properties hold:

- *Reflexivity:* A type is its own subtype since an occupied space includes itself.
- *Antisymmetry:* For aspects refined by overapproximation holds that if the space occupied according to T' is

geometrically included in the one of T but not identical, then there is at least a point in space that is occupied by T but not by T'. Thus, the space of T is not included in the one of T' and, in consequence, T is not a subtype of T' with respect to overapproximation. The argumentation for underapproximation is analog.

- *Transitivity:* If T' is a subtype of T and T'' a subtype of T' with respect to overapproximation, then the occupied space according to T'' is included in the one defined by T' and that one is included in the one according to T. Thus, the occupied space defined for T'' is also included in the one specified in T such that T'' is also a subtype of T. An analogous deduction can be drawn for underapproximation.

It is possible to construct a lattice based on this partial order for a fixed number of aspects. The type $\bot$ is a subtype of all other types. Here, all overapproximation-refinement aspects are occupying zero space all the time, while underapproximation-refinement aspects occupy all the space all the time. In contrast, all other types are subtypes of the $\top$ element. Thus, underapproximation-refinement aspects occupy zero space all the time, while overapproximation-refinement aspects occupy all the space all the time.

IV. Behavioral Type Checking and Evaluation

In this section, we discuss means to decide the compatibility of system components based on their behavioral types.

A. Type Compatibility Checking Algorithm

For two space-aware behavioral types with cycle times ct_1 and ct_2, we perform space-aware behavioral type checking in the following way:

1) We calculate the least common multiple of ct_1 and ct_2 that we name ct.
2) For all time points t between 0 and ct we perform the following steps:
   a) Retrieve for both behavioral types all relevant spatial information expressed by the behavioral descriptions bd_1 and bd_2 at timepoint t.
   b) Decide possible overlappings between the behavioral descriptions bd_1 and bd_2 by regarding the possibly occupied space for all underapproximation-refinement aspects. Here, an overlapping must occur, for each spatial aspect. Otherwise, the types are incompatible.
   c) Decide additional possible overlapping between spatial information of bd_1 and bd_2 by regarding the possibly occupied space for all overapproximation-refinement aspects. Here, no overlapping must occur for any spatial aspect. Otherwise, the types are incompatible.

The algorithm is carried out using the checker BeSpaceD [10] that, depending on the geometry used, converts the spatial information and property into a SAT or an SMT problem. For that, BeSpaceD breaks the geometric constraints down into more fine grained verification conditions as we discuss in the following.

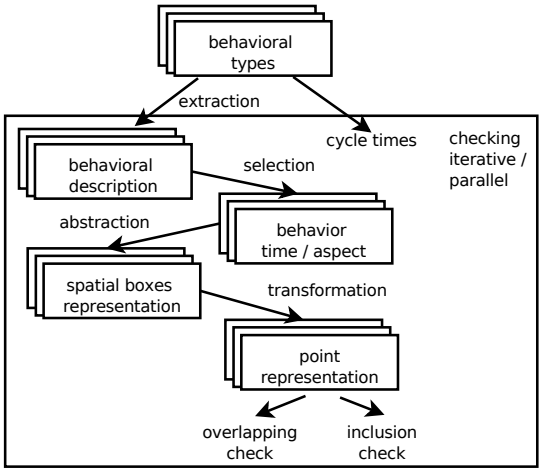

Fig. 4. Checking type compatibility and subtyping

B. Making Behavioral Descriptions Checkable

Our modeling style allows for very rich specifications describing quite complex systems. Checking these specifications would demand to treat a state space that would exceed the time and memory limits of the type checking algorithm introduced above. In the following, we present some steps allowing to abstract complex specifications into checkable ones such that our type compatibility checking and subtyping algorithms can be used. To guarantee that the abstractions do not falsify the verification results, they have to preserve the transitivity, reflexivity, and antisymmetry properties introduced in Sect. III. The abstraction consists of an order of operations that is depicted in Figure 4 (see also [10]):

1) *From time intervals to timepoints:* Time interval-based descriptions are transformed into timepoint-based descriptions by using safe approximations of geometric spatial behavior of adjacent time intervals at the timepoints.
2) *Extraction of relevant behavioral information:* BeSpaceD provides functions that are based on time and spatial aspects and provide sub-descriptions for the relevant behavior which are defined on the inductive structure of the behavioral descriptions.
3) *From segments to boxes:* Parts of robots may be described by segments or other geometric objects. Segments have a cylindric shape with a radius, a length, and an orientation. For fast and easy checking, we convert segments and other geometric objects into box-based approximations. Boxes are defined by an upper left front and a lower right rear coordinate that are both expressed by their respective x, y and z axes of the coordinate system. Figure 5 shows a variant of the second example from Sect. I-A in which the line representations of the three robot components are replaced by a number of boxes representing the space covered. As long as the

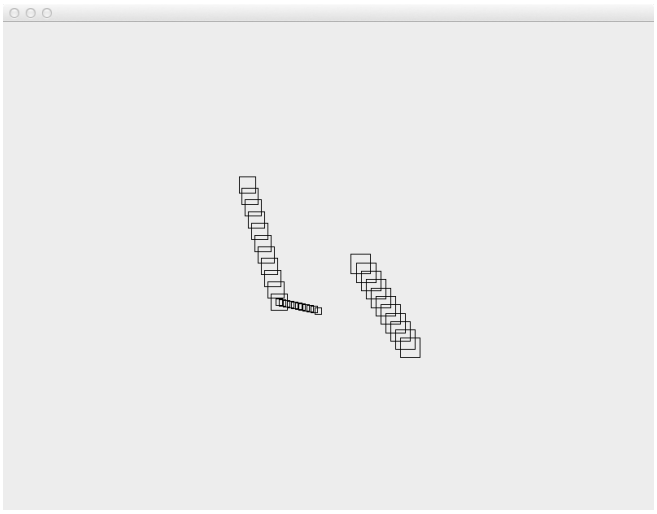

Fig. 5. Box-based abstraction of rotating robots

boxes cover all the space of the three components, this replacement is a safe overapproximation. (It would be a safe underapproximation if all space represented by the boxes was covered by the components.)

4) *Automata and spatial behavior:* The behavior of our components can be modeled using automata with a cyclic control flow. Here, we describe possible transitions and states encountered as events that are part of the behavioral description.
5) *From boxes to spacepoints:* Behavioral descriptions using geometric boxes can be broken down into descriptions that contain geometric points, so-called *spacepoints*. For example, a cube with a side length of 10 may be broken down into $10 \cdot 10 \cdot 10 = 1000$ spacepoints. In the behavioral description, each spacepoint is described using a predicate. In spite of this enlargement of the behavioral representation, we can check the spacepoints speedily since points from different behavioral descriptions are comparable without further interpretation.
6) *Checking of overlappings and inclusion with points:* We use hash-sets for checking overlappings and inclusion of two descriptions. For overlappings, we insert points from one description into the hash-set and check whether the points of the second description are already in the hash-set. For inclusion, we insert points from one description and check whether all points from the other description are indeed included in the hash-set.
7) *SMT and other approaches:* In addition to comparing geometric representations on a point level, we have developed SMT encodings of geometric objects that are more efficient for large sets of points [10]. Furthermore, checking of point-wise overlappings and inclusion can also be performed in BeSpaceD using a SAT solver.

C. Implementation

A first implementation of BeSpaceD and space-aware behavioral types exists. It is done in the functional programming language *Scala* which facilitates the break down and conversion of behavioral descriptions.

```
abstract class Invariant;

abstract class ATOM extends Invariant;

case class OR (t1 : Invariant, t2 : Invariant)
   extends Invariant;
case class AND (t1 : Invariant, t2 : Invariant)
   extends Invariant;
case class NOT (t : Invariant) extends Invariant;
case class IMPLIES (t1 : Invariant, t2 : Invariant)
   extends Invariant;
        ...
case class TimePoint [T](timepoint : T)
   extends ATOM;
case class TimeInterval [T]
 (timepoint1 : T, timepoint2 : T) extends ATOM;
case class Event[E] (event : E) extends ATOM;
        ...
case class Occupy3DBox
  (x1 : Int, y1: Int, z1 : Int,
   x2 : Int, y2 : Int, z2 : Int) extends ATOM;
case class OccupySegment3D
  (x1 : Int, y1 : Int, z1 : Int,
   x2 : Int, y2 :Int, z2 : Int, radius : Int)
   extends ATOM;
case class Occupy3DPoint (x:Int, y:Int, z: Int)
   extends ATOM
```

Fig. 6. Some Scala definitions

Behavioral descriptions are provided as abstract data types called `Invariant`. We chose this name since logical descriptions are supposed to capture the abstract behavior of a component during its entire lifetime. For look and feel, we provide an excerpt in Fig. 6. Some logical operators, predicates for time and events and geometric occupation of time are shown. The description language is more expressive than the subset used for space-aware behavioral types, e.g., time only needs to be a type with a partial order (parameter `T`) whereas in our semantics definitions above we used integers.

In the following, we discuss two features of the implementation:

Type system features: Using the type constructors above with the behavioral specifications, our type checking algorithm invoking the BeSpaceD tool allows us to check *(i)* space-aware behavioral type compatibility and *(ii)* whether a space-aware behavioral type is a subtype of another one. Note, that behavioral descriptions can look different, but may describe the same type. Our framework is able to decide both subtyping and type compatibility, since we exhaustively simulate possible behavior bounded by the a cycle time. In cases, where the behavioral descriptions use elements that we cannot check, we may still derive an order of types based on checkable spatial aspects. For all non-checkable aspects, we assume safe approximations. Hence, a type for which the behavioral specification is uncheckable for all aspects, is equivalent to $\bot$.

Speed of type checking: We implemented the space-aware behavioral types checking as described above. Checking can be done in acceptable time, e.g., checking two types with a cycle time of 1000 different timepoints and 15000 spacepoints

for the first resp. 20000 spacepoints for the second behavioral description was done in between seven and eight seconds on an Intel core i5 running 2.8 GHz with 8 GB RAM using Mac OS 10.8.4.

V. Conclusion

We presented behavioral types as a concept for space-aware systems facilitating the development, commissioning, maintenance, and refactoring of systems with cyber-physical characteristics. Using a robot system, we motivated, formally defined and discussed their applicability.

The approach is intended to be used in industrial automation. Facilities in the domain typically operate using cycles, after which behavior is repeated. For example, a robot in an assembly line may perform the same movement and operation on a workpiece over and over again with slight variations based on the color of a work piece. Our behavioral descriptions were designed with that kind of behavior in mind.

Moreover, we believe that the use of behavioral type-like specifications of cyber-physical systems is especially important for remote collaboration of engineering teams. Ongoing work in this direction comprises our collaborative engineering project [8] with a focus on remote handling of industrial installations in the Australian outback (such as mining sites) or for oil rigs.

References

[1] L. de Alfaro, T.A. Henzinger. Interface automata. Symposium on Foundations of Software Engineering, ACM , 2001.

[2] F. Arbab. Abstract Behavior Types: A Foundation Model for Components and Their Composition. Formal Methods for Components and Objects. vol. 2852 of LNCS, Springer-Verlag, 2003.

[3] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, Multiple viewpoint contract-based specification and design. Formal Methods for Components and Objects. Springer-Verlag, 2008.

[4] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier et al.. Contracts for System Design, INRIA, Rapport de recherche RR-8147, Nov. 2012.

[5] B. Bennett, A. G. Cohn, F. Wolter, M. Zakharyaschev. Multi-Dimensional Modal Logic as a Framework for Spatio-Temporal Reasoning. Applied Intelligence, Volume 17, Issue 3, Kluwer Academic Publishers, November 2002.

[6] J. O. Blech. Ensuring OSGi Component Based Properties at Runtime with Behavioral Types. Model-Driven Engineering, Verification, and Validation, 2013.

[7] J. O. Blech. Towards a Framework for Behavioral Specifications of OSGi Components. Formal Engineering approaches to Software Components and Architectures. Electronic Proceedings in Theoretical Computer Science, 2013.

[8] J. O. Blech, I. Peake, H. Schmidt, M. Kande, S. Ramaswamy, Sudarsan SD., and V. Narayanan. Collaborative Engineering through Integration of Architectural, Social and Spatial Models. *Emerging Technologies and Factory Automation (ETFA)*, IEEE, 2014.

[9] J. O. Blech and B. Schätz. Towards a Formal Foundation of Behavioral Types for UML State-Machines. UML and Formal Methods. Paris, France, ACM SIGSOFT Software Engineering Notes, August 2012.

[10] J. O. Blech and H. Schmidt. Towards Modeling and Checking the Spatial and Interaction Behavior of Widely Distributed Systems. Improving Systems and Software Engineering Conference, Melbourne, 2013.

[11] M. Bravetti, G. Zavattaro. A theory of contracts for strong service compliance. Mathematical Structures in Computer Science 19(3): 601–638, 2009.

[12] L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. Theoretical Computer Science, Elsevier, 2008.

[13] L. Caires and L. Cardelli.A Spatial Logic for Concurrency (Part I). Information and Computation, Vol 186/2 November 2003.

[14] L. Caires and L. Cardelli. A Spatial Logic for Concurrency (Part II). Theoretical Computer Science, 322(3) pp. 517–565, September 2004.

[15] L. Caires and H. Torres Vieira. SLMC: a tool for model checking concurrent systems against dynamical spatial logic specifications. Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2012.

[16] G. Castagna, N. Gesbert, L. Padovani. A theory of contracts for Web services. ACM Trans. Program. Lang. Syst. 31(5), 2009.

[17] S. Dal Zilio, D. Lugiez, C. Meyssonnier. A logic you can count on. Symposium on Principles of programming languages, ACM, 2004.

[18] J. L. Fiadeiro, A. Lopes. Consistency of Service Composition. Fundamental Approaches to Software Engineering (FASE), vol. 7212 of LNCS, Springer, 2012.

[19] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, O. Maler. SpaceEx: Scalable Verification of Hybrid Systems. Computer Aided Verification (CAV'11), 2011.

[20] S. Graf, R. Passerone, and S. Quinton. Contract-based reasoning for component systems with rich interactions. Embedded Systems Development, ser. Embedded Systems, vol. 20, pp. 139-154, Springer, 2014.

[21] R.H. Güting, R. Hartmut, and M. Schneider. Realm-based spatial data types: the ROSE algebra. The VLDB JournalThe International Journal on Very Large Data Bases 4.2 (1995): 243–286.

[22] F. Han, J. O. Blech, P. Herrmann, and H. Schmidt. Model-based Engineering and Analysis of Space-aware Systems Communicating via IEEE 802.11. In 39th Annual International Computers, Software & Applications Conference (COMPSAC), pages 638–646, IEEE Computer, 2015.

[23] F. Han, J. O. Blech, P. Herrmann, H. Schmidt. Towards Verifying Safety Properties of Real-Time Probabilistic Systems. Formal Engineering approaches to Software Components and Architectures, 2014.

[24] P. Herrmann, J.O. Blech, F. Han, H. Schmidt. A Model-based Toolchain to Verify Spatial Behavior of Cyber-Physical Systems. In 2014 Asia-Pacific Services Computing Conference (APSCC), IEEE Computer.

[25] D. Hirschkoff, É. Lozes, D. Sangiorgi. Minimality Results for the Spatial Logics. Foundations of Software Technology and Theoretical Computer Science, vol 2914 of LNCS, Springer, 2003.

[26] S. Kambhampati and L.S. Davis. Multiresolution path planning for mobile robots. Volume 2 , Issue: 3, Journal of Robotics and Automation, IEEE 1986.

[27] F. A. Kraemer and P. Herrmann. Automated Encapsulation of UML Activities for Incremental Development and Verification. In *Model Driven Engineering Languages and Systems (MoDELS)*, LNCS 5795, pages 571–585. Springer-Verlag, 2009.

[28] F. A. Kraemer, V. Slåtten and P. Herrmann. Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. *Journal of Systems and Software*, 82(12):2068–2080, 2009.

[29] J.-C. Latombe. Robot Motion Planning. Kluwer Academic Publishers, 1991.

[30] E.A. Lee, Y. Xiong. A behavioral type system and its application in ptolemy ii. Formal Aspects of Computing, 2004.

[31] B. Meyer. Applying "Design by Contract". Computer, 25, 10, pp. 40–51, IEEE, October 1992.

[32] P. Nuzzo, H. Xu, N. Ozay, J. Finn, A. Sangiovanni-Vincentelli, R. Murray, A. Donz, and S. Seshia, "A contract-based methodology for aircraft electric power system design," IEEE Access, vol. 2, pp. 1-25, 2014.

[33] L. Pasquale, C. Ghezzi, C. Menghi, Ch. Tsigkanos, and B. Nuseibeh. Topology aware adaptive security. In Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 43–48. ACM, 2014.

[34] A. Platzer. Differential dynamic logic for hybrid systems. Journal of Automated Reasoning, vol. 41.2: 143–189, Springer, 2008.

[35] A. Platzer, J.-D. Quesel. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). International Joint Conference on Automated Reasoning, pp. 171–178, LNCS 5195, Springer, 2008.

[36] P. Rigaux, M. Scholl, and Agnes Voisard. Spatial databases: with application to GIS. Morgan Kaufmann, 2001.

[37] M. Wenger, J. O. Blech and A. Zoitl. Behavioral Type-based Monitoring for IEC 61499. To appear in Emerging Technologies and Factory Automation (ETFA), IEEE, 2015.