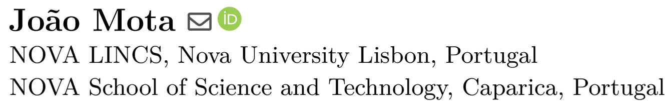
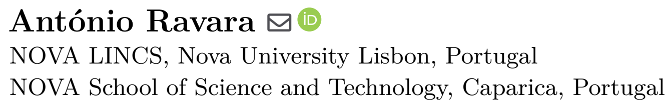
Behavioural Up/down Casting For Statically Typed Languages

Lorenzo Bacchiani
University of Bologna, Italy

Mario Bravetti
University of Bologna, Italy

Marco Giunti
University of Oxford, UK


João Mota
NOVA LINCS, Nova University Lisbon, Portugal
NOVA School of Science and Technology, Caparica, Portugal


António Ravara
NOVA LINCS, Nova University Lisbon, Portugal
NOVA School of Science and Technology, Caparica, Portugal

Abstract

We provide support for polymorphism in static typestate analysis for object-oriented languages with upcasts and downcasts. Recent work has shown how typestate analysis can be embedded in the development of Java programs to obtain safer behaviour at runtime, e.g., absence of null pointer errors and protocol completion. In that approach, inheritance is supported at the price of limiting casts in source code, thus only allowing those at the beginning of the protocol, i.e., immediately after objects creation, or at the end, and in turn seriously affecting the applicability of the analysis.

In this paper, we provide a solution to this open problem in typestate analysis by introducing a theory based on a richer data structure, named typestate tree, which supports upcast and downcast operations at any point of the protocol by leveraging union and intersection types. The soundness of the typestate tree-based approach has been mechanised in Coq.

The theory can be applied to most object-oriented languages statically analysable through typestates, thus opening new scenarios for acceptance of programs exploiting inheritance and casting. To defend this thesis, we show an application of the theory, by embedding the typestate tree mechanism in a Java-like object-oriented language, and proving its soundness.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Object oriented constructs; Theory of computation → Program verification

Keywords and phrases Behavioural types, object-oriented programming, subtyping, cast, typestates

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.5

Supplementary Material *Software (Coq Proofs Artifact)*: `https://zenodo.org/records/7712822`
Software (JaTyC Tool Artifact): `https://zenodo.org/records/7712915`
Software (JaTyC Tool on GitHub): `https://github.com/jdmota/java-typestate-checker`
archived at `swh:1:dir:69edd64b73a190021dd96ee97c7192722edfd00f`

Funding This work was partially supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233 (BehAPI).
Marco Giunti: EPSRC (EP/T006544/2).
João Mota: NOVA LINCS (UIDB/04516/2020) via the Portuguese Fundação para a Ciência e a Tecnologia (doi:10.54499/2021.05297.BD).

© Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 5; pp. 5:1–5:28

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Modern software engineering practices, e.g., Continuous Delivery [21], produce reliable software at high pace, through automatic pipelines of building, testing, etc. However, programming errors such as dereferencing null pointers [19] or using objects wrongly (e.g. reading from a closed file; closing a socket that timed out[1]) are often subtle and difficult to catch, even during the automated testing process. As put by Dijkstra [14]: *"program testing can be used to show the presence of bugs, but never to show their absence"*. So, tools to (statically) catch bugs are essential. Formal methods like deductive verification are difficult to adopt given the effort required [26], but lightweight static program analysis techniques can greatly improve the quality of the source code by detecting at compile-time logic errors, i.e., an unexpected action or behaviour. Beckman et al. [6] observe:

> In the open-source projects in our study [...] approximately 7.2% of all types defined protocols, while 13% of classes were clients of types defining protocols. [...] This suggests that protocol checking tools are widely applicable.

To tackle the challenge of finding bugs in object-oriented code, where objects naturally have protocols, in this paper we provide a protocol checking approach, supported by a tool, based on typestates [31, 15]. The work we present is applicable to most object-oriented languages, following the approach in closely related work [18, 11]: attach protocols (essentially, allowed orders of method calls) to classes and type check classes (i.e., their method bodies) following the protocol, thus gaining typestate-based nullness checking (ensuring memory-safety), protocol compliance, and protocol completion (under program termination).

In our previous work [3], we applied the approach to Java, proposing the JaTyC tool, exploiting the seminal simulation-based notion of subtyping [16] to check that the protocol of a class was a subtype of the protocol of its superclass. However, only upcasts and downcasts at the beginning of an object protocol (i.e., just after object creation) or at the end (i.e., in the `end` state) were allowed. Additionally, to determine if a typestate was a subtype of another, the simulation was only applied to the initial typestates of the protocols. It is crucial to overcome these limitations to make JaTyC applicable to real-world scenarios since, as shown in the study of Mastrangelo et al. [25], *casts are widely used*. The type checker was developed following a research methodology based on an iterative/incremental approach (see figure in **Appendix A** for details), based on the theory, which together with motivating examples, drove the type checker implementation (built upon the Checker Framework [30]).

Running example. To emphasise the relevance of our contribution, consider an example inspired from the automotive sector where driving dynamics control allows to customise the drive mode[2]; for SUVs, in particular, we consider a Comfort and a Sport modalities, where each allows specific features: EcoDrive and FourWheelsDrive, respectively.[3] List. 1 and List. 2 describe the behaviours of the controllers of a `Car` and a `SUV`, respectively, where class `SUV` extends `Car`. All cars have two base states: `OFF`, which models a powered off car, and `ON`, which represents a powered on car that can perform certain actions, e.g., set a concrete speed. In `OFF`, it is possible to `turnOn` the car and then access features like `setSpeed`. Dually, in `ON`, it is possible to `turnOff` the car. The `turnOn` action may, by some

[1] `https://github.com/redis/jedis/issues/1747`.
[2] BMW Sport vs Comfort modes: `bmwofstratham.com/bmw-sport-mode-vs-comfort-mode-stratham-nh`
[3] Code online: `github.com/jdmota/java-typestate-checker/tree/master/examples/car-example`

Listing 1 `Car` protocol.

```
typestate Car {
  OFF = {
    boolean turnOn():
      <true:ON,false:OFF>,
    drop: end
  }
  ON = {
    void turnOff(): OFF,
    void setSpeed(int): ON
  }
}
```

Listing 2 `SUV` protocol (`SUV` extends `Car`).

```
typestate SUV {
  OFF = {
    boolean turnOn(): <true:COMF_ON,false:OFF>,
    drop: end
  }
  COMF_ON = {
    void turnOff(): OFF,
    void setSpeed(int): COMF_ON,
    Mode switchMode(): <SPORT:SPORT_ON,COMFORT:COMF_ON>,
    void setEcoDrive(boolean): COMF_ON
  }
  SPORT_ON = {
    void turnOff(): OFF,
    void setSpeed(int): SPORT_ON,
    Mode switchMode(): <SPORT:SPORT_ON,COMFORT:COMF_ON>,
    void setFourWheels(boolean): SPORT_ON
  }
}
```

technical reason, fail, and so, depending on the returned value, either the resulting case is `ON` or `OFF`. SUVs are described by the protocol in Listing 2: when they are successfully powered on by means of `turnOn`, they are set in Comfort mode (`COMF_ON`), and in turn they enjoy specific operations, e.g., `setEcoDrive`. The mode can be changed by executing `switchMode`, whose result depends on the reached mode being still Comfort (as the operation may fail, e.g., if the speed is too high), or Sport (`SPORT_ON`). Similarly, the Sport mode provides the `switchMode` actions and also specific ones, e.g., `setFourWheels`. Note that `setSpeed` is overridden in the `SUV` class: if eco-drive is active, the speed must respect a given threshold, otherwise it can be set to any value. As we will see, in Section 6, overriding correctness is checked based on typestate variance, thus dynamic dispatch is guaranteed to work safely. Section 8 explains how our work compares with others dealing with inheritance.

Each protocol is defined by a set of *typestates* (e.g., in List. 1, `OFF` and `ON`), each one defining a set of callable methods and subsequent states, possibly depending on return values: e.g., if `turnOn` returns `true` in state `OFF` of the SUV protocol, then the next state is `COMF_ON`. By applying the subtyping algorithm by Gay and Hole [17] to the initial typestates (i.e., `OFF` in Car and SUV protocols), we see that the `SUV` protocol is a subtype of the `Car` one.

Listing 3 `upcast/downcast limitation` protocol.

```
public static void dispatch(@Requires("ON") Car c) { ··· }
public static void providePoweredSUV(@Requires("OFF") SUV c) {
    if (c.turnOn()) dispatch(c); // Upcast rejected by current typestate analysis
}
```

Key insight. Even for simple classes as `Car` and `SUV`, limiting casts only at the beginning/end of the protocols seriously reduces the programs we are able to typestate-check, such as the one in List. 3, where an automotive system dispatches already powered on cars (i.e., required in typestate `ON`), whether they are SUVs or not. Removing this limitation is challenging. The solution relies on the key insight that one has to run the subtyping algorithm not only on the pair of initial typestates, but on all pairs, to find all typestates in both protocols that are in a subtyping relation. For example, the `limitSpeed` method in List. 4 expects a `Car` in typestate `ON`. Since `SPORT_ON` is a subtype of `ON`, code passing a `SUV` in typestate `SPORT_ON` to `limitSpeed` is type-safe. However, if we run the subtyping algorithm starting from the pair of initial typestates of the given protocols (i.e., `(OFF,OFF)`), the generated simulation relation [4, 17] (in Figure 1, where boxes represent input states, and diamonds output ones), will not include `(SPORT_ON,ON)` (leftmost graph). If we provide `(SPORT_ON,ON)`, we realise that this pair is in the *typestate subtyping* relation (rightmost graph).

Listing 4 Limitation of the Subtyping Algorithm Application.

```
1 void limitSpeed(@Requires("ON") Car c, int speed) {
2     c.setSpeed(Math.min(speed, 50));
3 }
```

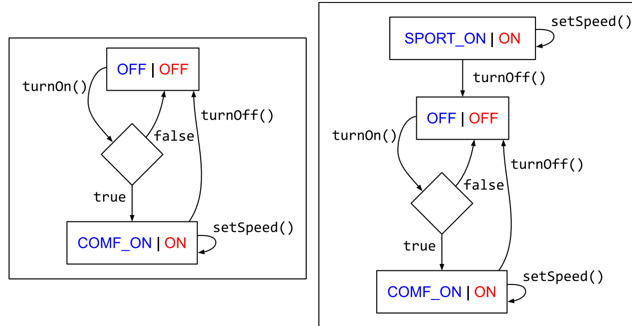


Figure 1 Subtyping simulations starting with `(OFF,OFF)` (left) and `(SPORT_ON,ON)` (right). Blue depicts typestates of the `SUV` protocol and red those of `Car`.

A theory of typestate upcast and downcast. With this key insight, we devise the following mechanism: when downcasting, we look for the typestates (in the protocol of the target class) that are subtypes of the current one; when upcasting, we look for the typestates that are supertypes of the current one. Since multiple typestates may be found, we need a structured notion of *types* to combine them. When downcasting, we combine the subtypes in a *union type* [5, 29] (modelling the fact that the actual type is unknown) so that a method call is allowed only if it is permitted by both elements of the union. Union types are also useful to allow branching code to be typed with different types so that subsequent code, complying with either branch, is accepted (e.g., after an `if` statement). This is more flexible than some other approaches (like session types ones [32]), which require both branches to have the same type. When upcasting, dually, we combine the supertypes in an *intersection type* so that a method call is allowed if it is permitted by at least one element of the intersection.

However, we need more than just intersection and union types. To illustrate the problem, consider a class for an Electric Car (`ECar`) which also extends `Car`. Consider the code in List. 5. After the `if` statement, is `c` an instance of `SUV` or `ECar`? Because of these scenarios, we associate classes with *types* and track all possibilities. To that end, we introduce *typestate trees*, which resemble the class hierarchy. Herein, the typestate tree would have a root for class `Car`, with child nodes for `SUV` and `ECar`. Each node corresponds to a class and maps to the type of the object, accounting for the case in which the object is indeed an instance of that class. In this way, in case of a future downcast to the `SUV` or `ECar` classes, we just consider the corresponding subtree corresponding to that class.

Listing 5 Typestate Tree motivation.

```
1 Car c;
2 if (cond) c = new SUV();
3 else c = new ECar();
```

Discussion. The solution we devise is language agnostic, applicable to many object-oriented languages. To test its expressiveness, we applied it to Java, extending JaTyC to now support (up/down)casting in the middle of a protocol. By doing this, we advance related work (Section 8). Kouzapas et al. [24] mention, in the future work section, that to cope

with protocol inheritance between two classes one just needs “a subtyping relation between their typestate specifications”. This is enough if one is only concerned with extending class inheritance but, as we showed, is not adequate to deal with casting, a very common feature.

Furthermore, we support *droppable typestates* (see typestate `OFF` in List. 1), the final typestates of a protocol – where one can either safely stop using the protocol or perform more actions (if there are any). A droppable typestate with no actions is similar to the `end` state in session types. To fully support droppable typestates, we provide a definition of subtyping over these, extending Gay and Hole's session type subtyping definition.

Contributions. In short, the main contributions of this work are:

- sound support for safely performing **upcasts** and **downcasts** at any point of a protocol (assuming class downcasts are performed to a class of which an object is a subtype of);
- formalisation of **subtyping over droppable typestates** (generalising Gay and Hole's session type subtyping);
- **mechanisation** of all definitions and proofs of our results in Coq (artifact available);
- implementation of the presented concepts in our **type checker for Java, JaTyC**, where we successfully run all examples included in this paper.

Advances with respect to the state of the art. As far as we know, no previous work deals with casts in the middle of protocols. Moreover, droppable states allow to mark states when the protocol can be safely stopped, another key concept. So, our work advances the state of the art with expressive support for inheritance and casting in object-oriented languages, leveraging on behavioural types [2, 22].

Structure of the paper. **Section 2** presents the subtyping relation, shown to be a pre-order, and a complete and sound algorithm to check typestates subtyping (Theorems 8 and 9). **Section 3** presents upcast and downcast, and the crucial result that each operation preserves subtyping (Theorems 23 and 28). Moreover, we show that, as expected, each operation reverses the other (Corollaries 29 and 30). Finally, we show that method calls make the typestates evolve, preserving subtyping (Theorem 33), and evolution on types commutes with upcast and downcast (Theorems 34 and 35). **Section 4** presents typestate trees, the crucial structure to allow up/down-casting in the middle of a protocol, and main results (Theorem 46 and 51). **Section 5** presents a key result to safely equip a programming language with our subtypestate mechanism – operations on typestate trees preserve their soundness (Theorem 54). **Section 6** discusses how to safely develop a type checking system with typestate trees. Notice that the main contribution of this paper is the provably safe subtypestates theory, paving the way to its use in (most) object-oriented languages. **Section 7** explains how the example presented in List. 6 is type checked with our tool and describes a suite of examples showing the expressiveness of our approach. **Section 8** discusses related work. **Section 9** concludes by envisioning future challenges, e.g., the mechanisation of a type(state tree)-safe object calculus with inheritance to use as basis for our Java implementation. **Appendix A** provides insights on the research methodology we adopted, while **Appendix B** provides a glossary listing all the notations used in the paper.

2 Types and subtyping

In this section, we present the types one can assign to terms of an object-oriented language taken into account, and the corresponding subtyping relations. We first describe typestates, which encode the current state of an object and specify the available methods (Section 2.1).

Then, we compose these in union and intersection types (Section 2.2). Unions track the possible typestates an object might be in, and intersections combine behaviour of two distinct typestates. These will be important when we present how casting works (Section 3).

2.1 Typestates

The following grammar (Def. 1) defines our typestates language. It is very similar to the one presented by Bravetti et al. [11]. The meta-variable m ranges over the set of *method identifiers* **MNames**, o ranges over the set of *output values* **ONames**, and s ranges over the set of *typestate names* **SNames**. The wide tilde stands for a sequence of values.

States are basically of two forms: input and output states. *Input states* $_d\{\widetilde{m : w}\}$ denote a set like $\{m_1{:}w_1, m_2{:}w_2, \ldots, m_n{:}w_n\}$ offering methods (being $n \geq 0$ a natural number), seen as input actions (i.e., external choices), followed by arbitrary states; the meaning is that by selecting method m_i, the input state transitions to state w_i. Input states may optionally be marked as *droppable* (with the subscript `drop` at the left of the set). This marks which input states are final. For example, in List. 1, the typestate `OFF` is defined as a droppable input state (which in the user defined protocol associated with the Java code is represented by the `drop:end` option). *Output states* $\langle\widetilde{o : u}\rangle$ denote a set like $\langle o_1{:}u_1, o_2{:}u_2, \ldots, o_n{:}u_n\rangle$, presenting all possible outcomes of a method call (values o_1 to o_n, being n a positive natural number), seen as output actions (i.e., internal choices), followed by input states or typestate names. We only consider boolean and enumeration values as outputs.

To deal with recursive behaviour, protocols use equational definitions of typestates.

▶ **Definition 1.** *Typestates, ranged over by meta-variable u, are terms generated by the following grammar. States are terms ranged over by meta-variable w.*

$$\begin{array}{lcl@{\qquad}lcl} u & ::= & _d\{\widetilde{m : w}\} \mid s & d & ::= & \varepsilon \mid \texttt{drop} \\ w & ::= & u \mid \langle\widetilde{o : u}\rangle & E & ::= & s = {}_d\{\widetilde{m : w}\} \end{array}$$

We assume that in $\langle\widetilde{o : u}\rangle$ we have at least one output, while in $_d\{\widetilde{m : w}\}$, we can have no inputs: $_{\texttt{drop}}\{\}$ represents the protocol ending state, also denoted by `end`. Moreover, in an equation E, typestate names s do not occur unguarded (i.e., we disregard equations like $s = s'$). We write $w^{\widetilde{E}}$ to denote state w associated with a set of defining equations. Therefore, we assume that each typestate name s that is used in w and in the body of equations $\widetilde{E}$ has a unique defining equation in $\widetilde{E}$. Let $\mathcal{W}$ be the set of terms $w^{\widetilde{E}}$ and $\mathcal{U}$ be the set of terms $u^{\widetilde{E}}$. Furthermore, let $\mathcal{X}$ be the subset of $\mathcal{W}$ containing only input states $_d\{\widetilde{m : w}\}$ and $\mathcal{Y}$ be the subset of $\mathcal{W}$ with only output states $\langle\widetilde{o : u}\rangle$. Meta-variables x and y range over $\mathcal{X}$ and $\mathcal{Y}$, respectively. Hereafter, whenever the finite set of equations $\widetilde{E}$ is clear from the context, we consider states w implicitly associated with $\widetilde{E}$. Moreover, we omit writing ε.

We can use the grammar introduced in Def. 1 to formally specify *protocols* associated to classes. A protocol is represented by $s^{\widetilde{E}}$, with s being the initial typestate name. For example, the protocol associated with class `Car` (List. 1) is $\texttt{OFF}^{E_{\texttt{Car}}}$ with $E_{\texttt{Car}}$ being:

$$\begin{array}{lcl} \texttt{OFF} & = & _{\texttt{drop}}\{\ \texttt{turnOn} : \langle\ \texttt{true} : \texttt{ON},\ \texttt{false} : \texttt{OFF}\ \rangle\ \} \\ \texttt{ON} & = & \{\ \texttt{turnOff} : \texttt{OFF},\ \texttt{setSpeed} : \texttt{ON}\ \} \end{array}$$

The `OFF` typestate is marked as *droppable* and offers a single method (i.e., `turnOn`) which, depending on the returned value (`true` or `false`), leads to either `ON` or `OFF`, respectively. The `ON` typestate offers two methods, `turnOff` and `setSpeed`, leading to `OFF` and `ON`, respectively.

State subtyping is key to support behavioural casting. In our setting, subtypes offer a superset of the supertype methods (input contravariance), and a subset of the supertype outputs (output covariance). To define it properly (with the intended properties), we follow

the work by Gay and Hole on session types subtyping [17]. Therefore, we define the subtyping relation as a simulation one (as protocols can be infinite state systems), and present a sound and complete algorithm to check if one state is a subtype of another. We first introduce function `def` to unfold typestate name definitions. The simulation relation follows.

▶ **Definition 2** (Typestate name definitions). *Function* $\texttt{def} : \mathcal{W} \to \mathcal{W} \setminus \mathbf{SNames}$ *is such that, given a state* $w^{\widetilde{E}} \in \mathcal{W}$*, if it is a typestate name,* $\texttt{def}(w^{\widetilde{E}})$ *yields the body of its defining equation; otherwise,* $\texttt{def}(w^{\widetilde{E}})$ *yields the given state* $w^{\widetilde{E}}$*. Formally,*

$$\texttt{def}(w^{\widetilde{E}}) = \begin{cases} x^{\widetilde{E}} & \textit{if } w^{\widetilde{E}} = s^{\widetilde{E'} \cup \{s=x\}} \textit{ for some } s, E', x \\ w^{\widetilde{E}} & \textit{otherwise} \end{cases}$$

▶ **Definition 3** (State simulation). *A relation* $R \subseteq \mathcal{W} \times \mathcal{W}$ *is a* state simulation*, if* $({w_1}^{E_1}, {w_2}^{E_2}) \in R$ *implies the following conditions:*

1. *If* $\texttt{def}({w_1}^{E_1}) = {}_{d_1}\{\widetilde{m : w}\}_1{}^{E_1}$ *then* $\texttt{def}({w_2}^{E_2}) = {}_{d_2}\{\widetilde{m : w}\}_2{}^{E_2}$ *and:*
   - **a.** *for each* $m'{:}w_2'$ *in* $\{\widetilde{m : w}\}_2$*, there is* w_1' *such that* $m'{:}w_1'$ *in* $\{\widetilde{m : w}\}_1$ *and* $({w_1'}^{E_1}, {w_2'}^{E_2}) \in R$.
   - **b.** *if* $d_2 = \texttt{drop}$ *then* $d_1 = \texttt{drop}$.
2. *If* $\texttt{def}({w_1}^{E_1}) = \langle \widetilde{o : u} \rangle_1{}^{E_1}$ *then* $\texttt{def}({w_2}^{E_2}) = \langle \widetilde{o : u} \rangle_2{}^{E_2}$ *and:*
   - **a.** *for each* $o'{:}u_1$ *in* $\langle \widetilde{o : u} \rangle_1$*, there is* u_2 *such that* $o'{:}u_2$ *in* $\langle \widetilde{o : u} \rangle_2$ *and* $({u_1}^{E_1}, {u_2}^{E_2}) \in R$.

Now we define subtyping, following standard approaches.

▶ **Definition 4** (Subtyping on states). *We say* w_1 *is a* subtype *of* w_2*, i.e.,* ${w_1}^{E_1} \leq_S {w_2}^{E_2}$*, if and only if there exists a state simulation* R *such that* $({w_1}^{E_1}, {w_2}^{E_2}) \in R$.

An example of a state simulation (Def. 3) follows (also depicted in rightmost graph of Figure 1). It is then easy to check, using Definition 4, that $\texttt{SPORT_ON}^{E_{\textsf{SUV}}} \leq_S \texttt{ON}^{E_{\textsf{Car}}}$.

$$\{(\texttt{SPORT_ON}^{E_{\textsf{SUV}}}, \texttt{ON}^{E_{\textsf{Car}}}), (\texttt{OFF}^{E_{\textsf{SUV}}}, \texttt{OFF}^{E_{\textsf{Car}}}),$$
$$(\langle \texttt{true} : \texttt{COMF_ON}, \texttt{false} : \texttt{OFF} \rangle^{E_{\textsf{SUV}}}, \langle \texttt{true} : \texttt{ON}, \texttt{false} : \texttt{OFF} \rangle^{E_{\textsf{Car}}}),$$
$$(\texttt{COMF_ON}^{E_{\textsf{SUV}}}, \texttt{ON}^{E_{\textsf{Car}}})\}$$

Notice that the common rule for session type subtyping of `end` states (i.e., $\texttt{end} \leq_S \texttt{end}$) is derivable from the previous definitions by just picking the relation $R = \{({}_{\texttt{drop}}\{\}, {}_{\texttt{drop}}\{\})\}$ and observing that it is a state simulation (Def. 3), thus ${}_{\texttt{drop}}\{\} \leq_S {}_{\texttt{drop}}\{\}$ holds by Def. 4.

As a sanity check, we show basic subtyping properties on states: reflexivity and transitivity.

▶ **Lemma 5** (Reflexivity). *For all* w*, then* $w \leq_S w$.

▶ **Lemma 6** (Transitivity). *For all* ${w_1}^{E_1}$*,* ${w_2}^{E_2}$*,* ${w_3}^{E_3}$*, if* ${w_1}^{E_1} \leq_S {w_2}^{E_2}$ *and* ${w_2}^{E_2} \leq_S {w_3}^{E_3}$*, then also* ${w_1}^{E_1} \leq_S {w_3}^{E_3}$.

Defining an algorithm to check state subtyping is crucial, not only because it shows that subtyping is decidable, but also for implementing a type checking procedure (Def. 7). To obtain an algorithm for checking state subtyping, we guarantee termination by always applying Assump, whenever applicable. The initial goal of the algorithm is the judgement $\varnothing \vdash {w_1}^{E_1} \leq_{S_{alg}} {w_2}^{E_2}$. This approach is similar to the session type subtyping algorithm presented by Gay and Hole [17]. We also show in Theorems 8 and 9 that the subtyping algorithm is complete and sound with respect to the coinductive definition $\leq_S$ (Def. 4).

▶ **Definition 7** (Algorithmic state subtyping). *The following inference rules define judgements* $\Sigma \vdash {w_1}^{E_1} \leq_{S_{alg}} {w_2}^{E_2}$ *in which* Σ *is a set of typestate pairs, containing assumed instances of the subtyping relation.*

$$\frac{({w_1}^{E_1}, {w_2}^{E_2}) \in \Sigma}{\Sigma \vdash {w_1}^{E_1} \leq_{S_{alg}} {w_2}^{E_2}} \textsc{Assump}$$

$$\frac{\begin{array}{c}\texttt{def}({w_1}^{E_1}) = {}_{d_1}\{\widetilde{m : w}\}_1{}^{E_1} \qquad \texttt{def}({w_2}^{E_2}) = {}_{d_2}\{\widetilde{m : w}\}_2{}^{E_2} \\ \forall\ m'{:}w_2' \in \{\widetilde{m : w}\}_2\ .\ \exists\ w_1'\ .\ m'{:}w_1' \in \{\widetilde{m : w}\}_1\ \wedge\ \Sigma, ({w_1}^{E_1}, {w_2}^{E_2}) \vdash {w_1'}^{E_1} \leq_{S_{alg}} {w_2'}^{E_2} \\ d_2 = \texttt{drop} \implies d_1 = \texttt{drop}\end{array}}{\Sigma \vdash {w_1}^{E_1} \leq_{S_{alg}} {w_2}^{E_2}} \textsc{Input}$$

$$\frac{\begin{array}{c}\texttt{def}({w_1}^{E_1}) = \langle\widetilde{o : u}\rangle_1{}^{E_1} \qquad \texttt{def}({w_2}^{E_2}) = \langle\widetilde{o : u}\rangle_2{}^{E_2} \\ \forall\ o'{:}u_1 \in \langle\widetilde{o : u}\rangle_1\ .\ \exists\ u_2\ .\ o'{:}u_2 \in \langle\widetilde{o : u}\rangle_2\ \wedge\ \Sigma, ({w_1}^{E_1}, {w_2}^{E_2}) \vdash {u_1}^{E_1} \leq_{S_{alg}} {u_2}^{E_2}\end{array}}{\Sigma \vdash {w_1}^{E_1} \leq_{S_{alg}} {w_2}^{E_2}} \textsc{Output}$$

▶ **Theorem 8** (Algorithm completeness). *If* ${w_1}^{E_1} \leq_S {w_2}^{E_2}$ *then* $\varnothing \vdash {w_1}^{E_1} \leq_{S_{alg}} {w_2}^{E_2}$*.*

▶ **Theorem 9** (Algorithm soundness). *If* $\varnothing \vdash {w_1}^{E_1} \leq_{S_{alg}} {w_2}^{E_2}$ *then* ${w_1}^{E_1} \leq_S {w_2}^{E_2}$*.*

2.2 Types

To statically track the possible typestates an object might be in, we combine them in union types. We also combine them in intersection types to describe combined behaviour from both typestates in the intersection. Their usefulness will be made clearer when we see the result of upcasting a type. Our type hierarchy is a lattice, thus supporting $\top$ and $\bot$ types. Note that *types* do not include class information. *Typestate Trees* will be used for that (Section 4).

▶ **Definition 10** (Types grammar). *We call types, ranged over by meta-variable* t*, the terms generated by the following grammar. Recall that* u *refers to typestate terms (Definition 1).*

$$t ::= t \cup t \mid t \cap t \mid u^{\widetilde{E}} \mid \top \mid \bot$$

For example, the union type $\texttt{COMF_ON}^{E_{\text{SUV}}} \cup \texttt{SPORT_ON}^{E_{\text{SUV}}}$ describes an object that might be in typestate `COMF_ON` or `SPORT_ON`.

Let $\mathcal{T}$ be the set of types produced by rule t. Now we need to define a subtyping notion to apply to types. The setting is inspired in work by Muehlboeck and Tate [27], in particular, their definition of reductive subtyping.

▶ **Definition 11** (Subtyping on types). *Let* $\leq\ \subseteq \mathcal{T} \times \mathcal{T}$ *be the relation defined by the following inductive rules.*

$$\frac{}{t \leq \top} \textsc{Top} \qquad \frac{}{\bot \leq t} \textsc{Bot} \qquad \frac{{u_1}^{E_1} \leq_S {u_2}^{E_2}}{{u_1}^{E_1} \leq {u_2}^{E_2}} \textsc{Typestates}$$

$$\frac{t \leq t_i}{t \leq t_1 \cup t_2} \textsc{Union_R}\ (i \in \{1,2\}) \qquad \frac{t_i \leq t}{t_1 \cap t_2 \leq t} \textsc{Intersection_L}\ (i \in \{1,2\})$$

$$\frac{t_1 \leq t \qquad t_2 \leq t}{t_1 \cup t_2 \leq t} \textsc{Union_L} \qquad \frac{t \leq t_1 \qquad t \leq t_2}{t \leq t_1 \cap t_2} \textsc{Intersection_R}$$

As a sanity check, we show basic subtyping properties on types: reflexivity and transitivity.

▶ **Lemma 12** (Reflexivity). *For all* t*, then* $t \leq t$*.*

▶ **Lemma 13** (Transitivity). *For all t, t', t'', if $t \leq t'$ and $t' \leq t''$, then $t \leq t''$.*

An algorithm to check that two types are in a subtyping relationship (i.e., $t \leq t'$) can be implemented by proof search on the inference rules in Def. 11. For these, one can observe that the combined syntactic height of the two types being tested always decreases [27]. Therefore, every recursive search path is guaranteed to always reach a point in which both types being compared are typestates $u \in \mathcal{U}$. Since the algorithm to test ${u_1}^{E_1} \leq_S {u_2}^{E_2}$ terminates, the overall algorithm to check subtyping also terminates. For example, it is easy to check that $\texttt{COMF_ON}^{E_{\text{SUV}}} \leq \texttt{COMF_ON}^{E_{\text{SUV}}} \cup \texttt{SPORT_ON}^{E_{\text{SUV}}}$, using the UNION_R rule in Def. 11.

3 Basic operations on types

In this section, we start by describing some preliminary assumptions on the hierarchy of classes, and then proceed to present the three main operations on types performed during type checking: `upcast` (Section 3.1), `downcast` (Section 3.2), and `evolve` (Section 3.3). To showcase these, we use the code in List. 6 that creates an object of type `SUV`, calls the method `turnOn`, switches mode and finally passes the object to method `setSpeed` (lines 3-6).

Listing 6 `ClientCode` class.

```
1  public class ClientCode {
2    public static void example() {
3      SUV suv = new SUV();
4      while (!suv.turnOn()) { System.out.println("turning on..."); }
5      suv.switchMode();
6      setSpeed(suv);
7    }
8    private static void setSpeed(@jatyc.lib.Requires("ON") Car car) {
9      if (car instanceof SUV && ((SUV) car).switchMode() == Mode.SPORT)
10       ((SUV) car).setFourWheels(true);
11     car.setSpeed(50);
12     car.turnOff();
13   }
14 }
```

The method `setSpeed` takes a `Car` in the `ON` state, enforced by the `@Requires` annotation (line 8). The behaviour provided by the `ON` state is also available in `COMF_ON` and `SPORT_ON`, so the method should be prepared to work with a `Car` in the `ON` state or a `SUV` (in `COMF_ON` or `SPORT_ON`). The method tests if the car is a `SUV` and tries to switch to the sport mode (line 9); if it succeeds, it proceeds to set the four wheels drive (line 10). Then, it sets the speed to a given value (line 11) and finishes the protocol by turning off the car (line 12).

Throughout this paper, $\mathcal{C}$ is the set of class names and c is a meta-variable ranging over its elements. Additionally, assume all classes belong to a single-inheritance hierarchy associated.

▶ **Definition 14** (Super relation on classes). `Super` *is a partial function such that, given a class c,* `Super`(c) *is the unique direct super class of c, if there is one.*

▶ **Definition 15** (Subtyping relation on classes). *The relation $\leq_C \subseteq \mathcal{C} \times \mathcal{C}$ is the reflexive and transitive closure of the* `Super` *relation.*

With classes and their `Super` relation, we now need to map classes to their corresponding protocols, containing only useful states (i.e., reachable states from the initial one).

▶ **Definition 16** (Reachable states). *The* immediate *state reachability relation is a relation over $\mathcal{W} \times \mathcal{W}$, defined as follows: $w'^{\widetilde{E}}$ is immediately reachable from $w^{\widetilde{E}}$, if:*

1. $w = {}_d\{\widetilde{m : w}\}$ *and* $\exists\ m'\ .\ m'{:}w'$ *in* ${}_d\{\widetilde{m : w}\}$*;*
2. $w = \langle \widetilde{o : u} \rangle$ *and* $\exists\ o'\ .\ o'{:}w'$ *in* $\langle \widetilde{o : u} \rangle$*;*
3. $w = s$ *and* $\widetilde{E}$ *includes the equation* $s = w'$*.*

The state reachability relation *is the reflexive and transitive closure of the immediate state reachability relation.*

Recall that each class c has an associated protocol $s^{\widetilde{E}}$, where s is its initial typestate name. We enforce that for any classes c and c' such that $\texttt{Super}(c') = c$, the protocols of c and c' are subtypes (i.e., the initial typestate of c' is a subtype of the initial typestate of c).

▶ **Definition 17** (Protocol input states). $\texttt{ProtInputs}(c)$ *is the set of all input states* $_d\{\widetilde{m : w}\}$ *that are reachable from protocol* $s^{\widetilde{E}}$ *of class* c*.*

By only considering reachable input states from the initial typestate name of the protocol, we perform an optimisation that avoids dealing with useless typestates.

To refer to the typestates occurring in a type, we introduce a dedicated auxiliary function.

▶ **Definition 18** (Typestates in a type). *Function* $\texttt{typestates} : \mathcal{T} \to \mathcal{P}(U)$ *is such that, given a type* $t \in \mathcal{T}$*,* $\texttt{typestates}(t)$ *yields the set of typestates occurring in* t*. Formally,*

$$\texttt{typestates}(t) = \begin{cases} \texttt{typestates}(t_1) \cup \texttt{typestates}(t_2) & \textit{if } t = t_1 \cup t_2 \textit{ or } t = t_1 \cap t_2 \\ \{t\} & \textit{if } t \in \mathcal{U} \\ \{\} & \textit{if } t = \top \textit{ or } t = \bot \end{cases}$$

3.1 Upcast

To upcast a typestate from class c to class c', we take all typestates in the protocol of c' that are supertypes of the original typestate, and combine them in an intersection type, combining behaviour from different types. If no supertypes are found, the "empty intersection" yields $\top$, signalling an error.[4] Since we take supertypes, upcast builds a new type that is a supertype of the original (guaranteed by Theorem 21); and because we intersect the supertypes, we build the most "precise" type possible with typestates in c' (guaranteed by Theorem 22).

▶ **Definition 19** (Upcast on types). *Function* $\texttt{upcast} : \mathcal{T} \times \mathcal{C} \times \mathcal{C} \to \mathcal{T}$ *is such that, given a type* t*, a class* c *whose protocol the typestates in* t *belong to, and a class* c' *we want to upcast to;* $\texttt{upcast}(t, c, c')$ *yields the type obtained by taking the intersection of all supertypes (in the protocol of class* c'*) of typestates included in* t*. The domain of* upcast *only includes triples* (t, c, c') *such that* $\texttt{typestates}(t) \subseteq \texttt{ProtInputs}(c)$ *and* $c \leq_C c'$*. Formally,*

$$\texttt{upcast}(t, c, c') = \begin{cases} \texttt{upcast}(t_1, c, c') \cup \texttt{upcast}(t_2, c, c') & \textit{if } t = t_1 \cup t_2 \\ \texttt{upcast}(t_1, c, c') \cap \texttt{upcast}(t_2, c, c') & \textit{if } t = t_1 \cap t_2 \\ \bigcap\{u' \in \texttt{ProtInputs}(c') \mid t \leq u'\} & \textit{if } t \in \mathcal{U} \\ t & \textit{if } t = \top \textit{ or } t = \bot \end{cases}$$

To see how `upcast` works, consider the `setSpeed` call in List. 6. In line 6, after calling `switchMode`, the type of `suv` is $\texttt{COMF_ON} \cup \texttt{SPORT_ON}$ (since we ignore the output of `switchMode`, we do not know the actual typestate). To compute the type of the object passed as parameter, we use the `upcast` function, using as input: (*i*) $\texttt{COMF_ON} \cup \texttt{SPORT_ON}$ as the type to be upcast; (*ii*) `SUV` as the starting class; (*iii*) `Car` as the target class. Since the given type is a union type composed by two elements, the `upcast` function initially unfolds it and creates one intersection for each element (i.e., `COMF_ON` and `SPORT_ON`) containing all their supertypes. In this case, there is just one supertype for each: `ON`. Thus,

$$\texttt{upcast}(\texttt{COMF_ON} \cup \texttt{SPORT_ON}, \texttt{SUV}, \texttt{Car}) = \texttt{ON} \cup \texttt{ON} = \texttt{ON} \ .$$

[4] In general, upcast operations are always possible, since they produce a supertype of the original type. The issue here is that no operations are safely allowed on $\top$, so in practise, even if an error is not immediately reported on upcast, there will be an error when trying to use an object with $\top$ type.

As a sanity check, we show that `upcast` builds a type where the typestates composing it belong to the class we upcast to. Recall that Def. 19 has constraints $\texttt{typestates}(t) \subseteq \texttt{ProtInputs}(c)$ and $c \leq_C c'$ (the following results assume them). To improve readability we omit stating the constraints explicitly and simply quantify universally types and classes.

▶ **Lemma 20** (Upcast preserves protocol membership). *For all t, c and c', then*

$$\texttt{typestates}(\texttt{upcast}(t, c, c')) \subseteq \texttt{ProtInputs}(c').$$

To ensure `upcast` correctness, we show that the result: (i) is a supertype of the given type (Theorem 21); (ii) is the "closest" type to the original with typestates in the protocol of the target class (Theorem 22); and (iii) preserves the subtyping relation (Theorem 23), i.e., `upcast` on types in a subtyping relation produces types that are still in such relation.

▶ **Theorem 21** (Upcast Consistency). *For all t, c and c', we have $t \leq \texttt{upcast}(t, c, c')$.*

▶ **Theorem 22** (Upcast Least Upper Bound). *For all t, t', c and c', such that $\texttt{typestates}(t') \subseteq \texttt{ProtInputs}(c')$ and $t \leq t'$, we have $\texttt{upcast}(t, c, c') \leq t'$.*

▶ **Theorem 23** (Upcast Preserves Subtyping). *For all t, t', c and c', such that $t \leq t'$, we have $\texttt{upcast}(t, c, c') \leq \texttt{upcast}(t', c, c')$.*

3.2 Downcast

To downcast a typestate from class c to c', we take all typestates in the protocol of c' that are subtypes of the original typestate, and combine them in a union type. We use a union type because we need to account for all possible typestates an object might be in. Since we take the subtypes, downcast builds a new type that is a subtype of the original one (guarantee given by Theorem 26); and because we make the union of them, we build the "closest" type possible with typestates in c' (guarantee given by Theorem 27).

▶ **Definition 24** (Downcast on types). *Function `downcast` $: \mathcal{T} \times \mathcal{C} \times \mathcal{C} \to \mathcal{T}$ is such that, given a type t, the class c whose protocol the typestates in t belong to, and the class c' we want to downcast to; $\texttt{downcast}(t, c, c')$ yields the type obtained by taking the union of all subtypes (in the protocol of class c') of typestates included in t. The domain of `downcast` only includes triples (t, c, c') such that $\texttt{typestates}(t) \subseteq \texttt{ProtInputs}(c)$ and $c' \leq_C c$. Formally,*

$$\texttt{downcast}(t, c, c') = \begin{cases} \texttt{downcast}(t_1, c, c') \cup \texttt{downcast}(t_2, c, c') & \textit{if } t = t_1 \cup t_2 \\ \texttt{downcast}(t_1, c, c') \cap \texttt{downcast}(t_2, c, c') & \textit{if } t = t_1 \cap t_2 \\ \bigcup \{u' \in \texttt{ProtInputs}(c') \mid u' \leq t\} & \textit{if } t \in \mathcal{U} \\ t & \textit{if } t = \top \textit{ or } t = \bot \end{cases}$$

Note that `downcast` only yields $\bot$ if given $\bot$. Consider the third case of Def. 24. The union only yields $\bot$ if no sub-typestates in the protocol of c' are found. But that is impossible. If we downcast from a typestate t (in c) to a subclass c', and since the protocol of c' is a subtype of the one of c, there will necessarily be at least one typestate in c' subtype of t. Moreover, Theorem 54 will show that our overall approach is sound.

To see how `downcast` works, consider the downcast performed in line 9 of List. 6. To compute the type of `(SUV) car`, we use `downcast`, defined in Def. 24, passing as parameter: (i) `ON` as the type to be downcast (given the `Requires` annotation); (ii) `Car` as the starting class; (iii) `SUV` as the target class. Since the type passed as parameter is a simple typestate, the `downcast` function just creates a union containing all the subtypes of `ON`. Concretely,

$$\texttt{downcast}(\texttt{ON}, \texttt{Car}, \texttt{SUV}) = \texttt{COMF_ON} \cup \texttt{SPORT_ON}\ .$$

As a sanity check, we show that `downcast` builds a type where the typestates composing it belong to the class we downcast to. Recall that Def. 24 has constraints $\texttt{typestates}(t) \subseteq \texttt{ProtInputs}(c)$ and $c' \leq_C c$. (the following results assume them). To improve readability, the constraints are implicit and we simply quantify universally types and classes.

▶ **Lemma 25** (Downcast preserves protocol membership). *For all t, c and c', we have*

$\texttt{typestates}(\texttt{downcast}(t, c, c')) \subseteq \texttt{ProtInputs}(c')$.

To ensure `downcast` correctness, we show that the result: (i) is a subtype of the given type (Theorem 26); (ii) is the "closest" type to the original with typestates in the protocol of the target class (Theorem 27); and (iii) preserves the subtyping relation i.e., `downcast` on types in a subtyping relation produces types that are still in such relation (Theorem 28).

▶ **Theorem 26** (Downcast Consistency). *For all t, c and c', we have* $\texttt{downcast}(t, c, c') \leq t$.

▶ **Theorem 27** (Downcast Greatest Lower Bound). *For all t, t', c and c', such that* $\texttt{typestates}(t') \subseteq \texttt{ProtInputs}(c')$ *and $t' \leq t$, we have* $t' \leq \texttt{downcast}(t, c, c')$.

▶ **Theorem 28** (Downcast Preserves Subtyping). *For all t, t', c and c', such that $t \leq t'$, we have* $\texttt{downcast}(t, c, c') \leq \texttt{downcast}(t', c, c')$.

Additionally, we relate the result of upcasting and then downcasting with the original type, as well as, the result of downcasting and then upcasting. The first follows from Theorems 21 and 27, the second from Theorems 22 and 26. These corollaries are also important to ensure the soundness of the approach (Theorem 54).

▶ **Corollary 29** (Downcast reverses upcast). *For all t, c and c', we have*

$t \leq \texttt{downcast}(\texttt{upcast}(t, c, c'), c', c)$.

▶ **Corollary 30** (Upcast reverses downcast). *For all t, c and c', we have*

$\texttt{upcast}(\texttt{downcast}(t, c, c'), c', c) \leq t$.

3.3 Evolve

Whenever we perform a method call on an object with a given type, we need to compute the new type representing the typestates the object might be in after the call. To compute such type and rule out misconduct, we define the `evolve` function, which yields $\top$ when a method is not callable in the given type. $\texttt{Ret}_m$ is the set of outputs returnable by method m.

▶ **Definition 31** (Evolve). *Function* $\texttt{evolve} : \mathcal{T} \times \mathcal{M} \times \mathcal{O} \to \mathcal{T}$ *is such that, given a type t, a method m, and an object $o \in \texttt{Ret}_m$;* $\texttt{evolve}(t, m, o)$ *yields the new type obtained by executing m on any object currently with type t, where o is the value returned by m. Its definition relies on the auxiliary functions* $\texttt{evolveU} : \mathcal{U} \times \mathcal{M} \times \mathcal{O} \to \mathcal{U}$ *and* $\texttt{evolveY} : \mathcal{Y} \times \mathcal{O} \to \mathcal{U}$. *Formally,*

$$\texttt{evolve}(t, m, o) = \begin{cases} \texttt{evolve}(t_1, m, o) \cup \texttt{evolve}(t_2, m, o) & \text{if } t = t_1 \cup t_2 \\ \texttt{evolve}(t_1, m, o) \cap \texttt{evolve}(t_2, m, o) & \text{if } t = t_1 \cap t_2 \\ \texttt{evolveU}(t, m, o) & \text{if } t \in \mathcal{U} \\ t & \text{if } t = \top \text{or } t = \bot \end{cases}$$

$$\mathtt{evolveU}(u,m,o) = \begin{cases} w & \textit{if } \mathtt{def}(u) =_{d} \{m : w, \widetilde{m : w}\} \wedge w \in \mathcal{U} \\ \mathtt{evolveY}(w,o) & \textit{if } \mathtt{def}(u) =_{d} \{m : w, \widetilde{m : w}\} \wedge w \in \mathcal{Y} \\ \top & \textit{otherwise} \end{cases}$$

$$\mathtt{evolveY}(y,o) = \begin{cases} u & y = \langle o : u, \widetilde{o : u}\rangle \\ \bot & \textit{otherwise} \end{cases}$$

Since `evolve` is deterministic, it is defined as a function, not as a labelled transition system.

To see how `evolve` works, consider the `switchMode` call in line 9 of List. 6. To compute the type of `car`, we use `evolve`, defined in Def. 31, passing as parameter: (i) the type `COMF_ON` $\cup$ `SPORT_ON` be evolved (given the result of the `downcast` function); (ii) the method `switchMode` to make the type evolve; (iii) the expected output `Mode.SPORT` to enter the `if` branch. Since the type, passed as parameter, is a union type composed by two elements, the `evolve` function is called recursively, and then the auxiliary function `evolveU` is called for `COMF_ON` and `SPORT_ON`. Since the `switchMode` action leads to an output state, for both `COMF_ON` and `SPORT_ON` (see List. 2), the auxiliary function `evolveY` is invoked. Concretely,

$$\mathtt{evolve}(\mathtt{COMF_ON} \cup \mathtt{SPORT_ON}, \mathtt{switchMode}, \mathtt{Mode.SPORT}) = \mathtt{SPORT_ON} \cup \mathtt{SPORT_ON}$$

that can be simplified to `SPORT_ON`. Notice that the evolved type has the same structure of the one before calling the `evolve` function i.e., a union type.

As a sanity check, we show that `evolve` produces a type containing only typestates belonging to the initial class.

▶ **Lemma 32** (Evolve preserves protocol membership). *For all t, m, o, c,*

$$\mathtt{typestates}(t) \subseteq \mathtt{ProtInputs}(c) \textit{ implies } \mathtt{typestates}(\mathtt{evolve}(t,m,o)) \subseteq \mathtt{ProtInputs}(c)$$

To ensure `evolve` correctness, we show that the result preserves the subtyping relation: `evolve` on types in a subtyping relation produces types that still are in such relation.

▶ **Theorem 33** (Evolve preserves subtyping). *For all t and t' such that $t \leq t'$, we have that*

$$\mathtt{evolve}(t,m,o) \leq \mathtt{evolve}(t',m,o).$$

We also relate `evolve` with `upcast` and `downcast` showing that: (i) `upcast` after `evolve` produces a subtype of the inverse sequence of operations (Theorem 34); and (ii) `downcast` after `evolve` produces a supertype of the inverse sequence of operations (Theorem 35). These theorems are key to ensure soundness (Theorem 54). For readability, we omit the constraints on the universally quantified variables needed to use `upcast` and `downcast`.

▶ **Theorem 34** (Evolve and upcast). *For all t, m, o, c and c', we have that*

$$\mathtt{upcast}(\mathtt{evolve}(t,m,o),c,c') \leq \mathtt{evolve}(\mathtt{upcast}(t,c,c'),m,o).$$

▶ **Theorem 35** (Evolve and downcast). *For all t, m, o, c and c', we have that*

$$\mathtt{evolve}(\mathtt{downcast}(t,c,c'),m,o) \leq \mathtt{downcast}(\mathtt{evolve}(t,m,o),c,c')$$

4 Typestate Trees

In this section, we describe *Typestate Trees*, the crucial data structure we use to solve the problem of casting in the middle of a protocol. These trees associate classes with types containing only states in the protocol of those classes (i.e., $\texttt{typestates}(t) \subseteq \texttt{ProtInputs}(c)$). The tree root indicates the static type of a variable and the corresponding type (in $\mathcal{T}$) at a given program point. All other nodes describe what should be the type if we downcast to the corresponding class. The type in the root is always a sound approximation of the runtime execution. The types in other nodes are also sound only if the object is an instance of the corresponding class. This will imply that type safety is guaranteed up-to class downcasts being performed to a class of which an object is a subtype of. Hereafter we define well-formed typestate trees and auxiliary functions. Sections 4.1, 4.2, 4.3, and 4.4, describe the main operations on typestate trees: `upcastTT`, `downcastTT`, `evolveTT`, and `mrgTT`, respectively.

▶ **Definition 36** (Typestate Trees). *Recall that c ranges over classes ($\mathcal{C}$) and t ranges over types ($\mathcal{T}$). Let $\mathcal{TT}$ be the smallest set of triples satisfying the following rules:*

$$\frac{}{(c, t, \{\}) \in \mathcal{TT}} \qquad \frac{n \geq 1 \qquad \forall i, 1 \leq i \leq n \;.\; tt_i \in \mathcal{TT}}{(c, t, \{tt_i \,|\, 1 \leq i \leq n\}) \in \mathcal{TT}}$$

Notice that triples in $\mathcal{TT}$ represent trees and are composed of: the class c and the type t of the root, and a set of subtrees (again triples in $\mathcal{TT}$), one for each root child. Such a set is empty if the tree root has no children (i.e., the tree simply represents a leaf). Throughout this paper, tt ranges over elements of $\mathcal{TT}$ and tts ranges over sets of elements of $\mathcal{TT}$. We need functions to destroy an element of $\mathcal{TT}$ (which is a triple). Let $\texttt{cl}((c, t, tts)) = c$, $\texttt{ty}((c, t, tts)) = t$, and $\texttt{children}((c, t, tts)) = tts$.

▶ **Definition 37** (No duplicate classes). *The predicate* `nodup` *over $\mathcal{P}(\mathcal{TT})$ asserts that, given a set $tts \in \mathcal{P}(\mathcal{TT})$, no two typestates trees in tts have the same associated class. Formally,* $\texttt{nodup}(tts)$ *holds if:* $\forall\, tt, tt' \in tts \;.\; \texttt{cl}(tt) = \texttt{cl}(tt') \Rightarrow tt = tt'$.

▶ **Definition 38** (Well-formedness Of Typestate Trees). *The predicate $\vdash$ over $\mathcal{TT}$ asserts that, given a typestate tree (c, t, tts), it is correctly constructed. Formally,*

$$\frac{\begin{array}{c}\texttt{typestates}(t) \subseteq \texttt{ProtInputs}(c) \\ \texttt{nodup}(tts) \qquad \forall tt \in tts \;.\; \texttt{Super}(\texttt{cl}(tt)) = c \wedge \texttt{upcast}(\texttt{ty}(tt), \texttt{cl}(tt), c) \leq t \wedge \vdash tt\end{array}}{\vdash (c, t, tts)}$$

So, a typestate tree (c, t, tts) is well-formed under the following conditions: (i) all the typestates of type t belong to the protocol of class c; (ii) there are no two children with the same class; (iii) the classes associated with each child tree are direct subclasses of c; (iv) if we upcast a type of a child tree, we get a subtype of t; and (v) each child is also well-formed. Condition (iv) ensures that the type of a child tree is never less "precise" than the type of the parent. From now on, we only consider well-formed typestate trees.

To illustrate the concept, suppose that in line 3 of List. 6, instead of assigning the newly created object to a `SUV` variable, we assign it to a `Car` one, performing an upcast. Since the static and actual type are different, we need a typestate tree to handle future casts. Given Def. 36 and Def. 38, the resulting typestate tree is `(Car, OFF, {(SUV, OFF, {})})`.

4.1 Upcast

Upcasting a typestate tree to class c' ensures that the resulting root class is c', by recursively following the `Super` relation and building up new tree roots until the root class is c'.

▶ **Definition 39** (Upcast on typestate trees). *Function* upcastTT : $\mathcal{TT} \times \mathcal{C} \to \mathcal{TT}$ *is such that* upcastTT$((c, t, tts), c')$ *performs an upcast on typestate tree* (c, t, tts) *to class* c'. *The domain of* upcastTT *only includes pairs* $((c, t, tts), c')$ *such that* $c \leq_C c'$. *Formally,*

$$\texttt{upcastTT}((c, t, tts), c') =$$

$$\begin{cases} (c, t, tts) & \text{if } c = c' \\ \texttt{upcastTT}((\texttt{Super}(c), \texttt{upcast}(t, c, \texttt{Super}(c)), \{(c, t, tts)\}), c') & \text{otherwise} \end{cases}$$

Notice that, under the assumption on the domain of the upcastTT, the function terminates since the distance between c and c' decreases with each recursive step.

▶ **Theorem 40** (Typestate Trees Well-formedness Preserved By Upcast). *For all* c'', tt, *such that* $\vdash tt$ *and* $\texttt{cl}(tt) \leq_C c''$, *it holds that* $\vdash \texttt{upcastTT}(tt, c'')$.

To see how upcastTT works, consider the setSpeed call in List. 6. In line 8, after calling switchMode, the object suv has the following typestate tree $(\texttt{SUV}, \texttt{COMF_ON} \cup \texttt{SPORT_ON}, \{\})$. When passing suv to setSpeed, we need to upcast from SUV to Car. To do that, we use the upcastTT function, defined in Def. 39, passing as parameter: (i) $(\texttt{SUV}, \texttt{COMF_ON}\cup\texttt{SPORT_ON}, \{\})$ as the typestate tree to be upcast; and (ii) Car as the target class. Thus,

$$\texttt{upcastTT}((\texttt{SUV}, \texttt{COMF_ON}\cup\texttt{SPORT_ON}, \{\}), \texttt{Car}) = (\texttt{Car}, \texttt{ON}, \{(\texttt{SUV}, \texttt{COMF_ON}\cup\texttt{SPORT_ON}, \{\})\})\ .$$

It is crucial to notice that to upcast a typestate tree, we must perform multiple upcasts, incrementally building up new tree roots, not only to preserve the well-formedness property, but also to ensure soundness. For readability sake, we show the problem with an abstract, but simple example. Take classes A, B and C, where $\texttt{Super}(\texttt{C}) = \texttt{B}$, $\texttt{Super}(\texttt{B}) = \texttt{A}$, and the protocol equations associated with each class listed below. Recall that $\texttt{end} = {}_{\texttt{drop}}\{\}$.

$$\begin{array}{lll} \texttt{A1} & = & \{\ \texttt{m1} : \texttt{end}\ \} \\ \texttt{B1} & = & \{\ \texttt{m1} : \texttt{end},\ \texttt{m2} : \texttt{end}\ \} \end{array} \qquad \begin{array}{lll} \texttt{C1} & = & \{\ \texttt{m1} : \texttt{end},\ \texttt{m2} : \texttt{end},\ \texttt{m3} : \texttt{C2}\ \} \\ \texttt{C2} & = & \{\ \texttt{m1} : \texttt{end},\ \texttt{m4} : \texttt{end}\ \} \end{array}$$

Given the protocols above and according to Def. 4 we have:

$$\texttt{C1} \leq_S \texttt{B1} \leq_S \texttt{A1} \text{ and } \texttt{C2} \leq_S \texttt{A1}, \text{ but } \texttt{C2} \not\leq_S \texttt{B1}\ .$$

C2 not being a subtype of B1 is not a problem *per se*, but it may be when upcasting, if we define it to go directly to the root instead of going level-by-level, as downcasting after upcasting should lead to the original state.[5] To see that, consider the code in List. 7, which contains an unsafe method call, but would be accepted. At first, we create an object c of class C and we call its method m3, producing a new typestate, i.e., C2. Then, we assign c to variable a performing an upcast from class C to A (and from typestate C2 to A1). We finally perform a sequence of downcasts on a leading the object to class C (and to typestate C1).

Listing 7 Direct upcast example.

```
1 C c = new C(); // C1
2 c.m3(); // C2
3 A a = c; // A1: unsound upcast!
4 B b = (B) a; // B1: downcast level-by-level
5 C c = (C) b; // C1: incorrect! the state should be C2 (that of line 2)
6 c.m2(); // unsafe!
```

[5] Technically, downcasting after upcasting returns an over-approximation of the original state.

In detail (for those interested), notice that the result of upcasting C2 directly to class A (line 3, List. 7) is A1, since it is the only supertype of C2, i.e., $\texttt{C2} \leq_S \texttt{A1}$. To downcast A1 to class B, we check all the typestates in the protocol of B subtypes of A1. Since only B1 is subtype of A1, that is the downcast result (line 4). Similarly, since only C1 is a subtype of B1, it is the result of downcasting from B1 (line 5). Notice how a direct upcast to A, followed by a downcast to B, and then to C, results in a different typestate with respect to the initial one. This is unsound: C1 and C2 are unrelated. The issue is that a direct upcast to A makes us lose the information about C1 not having supertypes among typestates in B. Since we first upcast C2 to B, getting $\top$ as result, we find out that C2 has no supertypes among typestates in B. Additionally, since we use typestate trees, downcasting to C leads back to C2.

4.2 Downcast

When downcasting a given typestate tree tt to class c, we ensure that the root class of the resulting tree is c. If we find a subtree in tt whose class is c, we pick it as the result (by well-formedness, it is unique). Otherwise, we build a new tree downcasting from the most "precise" type information in tt. For this, we use the closestSubT function to look for the subtree whose class is hierarchically the "closest" to c.

▶ **Definition 41** (Closest subtree). *The function* $\texttt{closestSubT} : \mathcal{TT} \times \mathcal{C} \to \mathcal{TT}$ *is such that* $\texttt{closestSubT}(tt, c)$ *yields the subtree associated with the closest superclass of c occurring in tt. The domain of* closestSubT *only includes pairs* (tt, c) *such that* $c \leq_C \texttt{cl}(tt)$*. Formally,*

$$\texttt{closestSubT}(tt, c) = \begin{cases} \texttt{closestSubT}(tt', c) & \textit{if } c \leq_C \texttt{cl}(tt') \wedge tt' \in \texttt{children}(tt) \\ tt & \textit{otherwise} \end{cases}$$

To illustrate the use of closestSubT, consider classes A, B, and C, where $\texttt{Super}(\texttt{B}) = \texttt{A}$ and $\texttt{Super}(\texttt{C}) = \texttt{B}$. Let tt be $(\texttt{A}, t, \{(\texttt{B}, t', \{\})\})$. Then the following equalities hold: $\texttt{closestSubT}(tt, \texttt{A}) = tt$; $\texttt{closestSubT}(tt, \texttt{B}) = (\texttt{B}, t', \{\})$; and $\texttt{closestSubT}(tt, \texttt{C}) = (\texttt{B}, t', \{\})$. The first two cases are easy to understand: the function yields the subtree whose class is precisely the one we are looking for. In the third case, since there is no subtree in tt whose class is C, $\texttt{closestSubT}(tt, \texttt{C})$ yields the subtree corresponding to B, which is the "closest" superclass of C present in tt, i.e., $(\texttt{B}, t', \{\})$. Now, suppose instead that $\texttt{Super}(\texttt{B}) = \texttt{A}$ and $\texttt{Super}(\texttt{C}) = \texttt{A}$ (i.e., B and C are "siblings"). Then $\texttt{closestSubT}(tt, \texttt{C})$ would yield the entire tree tt whose class is A, which is the "closest" superclass of C present in tt. Lemma 42 ensures the correctness of closestSubT and is useful for the soundness proof (Theorem 54).

▶ **Lemma 42** (Closest correctness). *For all tt and c, if* $c \leq_C \texttt{cl}(tt)$ *then*

$$c \leq_C \texttt{cl}(\texttt{closestSubT}(tt, c)).$$

▶ **Definition 43** (Downcast on typestate trees). *Function* $\texttt{downcastTT} : \mathcal{TT} \times \mathcal{C} \to \mathcal{TT}$ *is such that* $\texttt{downcastTT}(tt, c)$ *performs a downcast on typestate tree tt to class c. The domain of* downcastTT *only includes pairs* (tt, c) *such that* $c \leq_C \texttt{cl}(tt)$*. Formally,*

$$\texttt{downcastTT}(tt, c) =$$

$$\begin{cases} tt' & \textit{if } tt' = \texttt{closestSubT}(c, tt) \wedge c = \texttt{cl}(tt') \\ (c, \texttt{downcast}(\texttt{ty}(tt'), \texttt{cl}(tt'), c), \{\}) & \textit{otherwise } tt' = \texttt{closestSubT}(c, tt) \end{cases}$$

▶ **Theorem 44** (Typestate Trees Well-formedness Preserved By Downcast). *For all c, tt, such that $\vdash tt$ and $c \leq_C \texttt{cl}(tt)$, it holds that $\vdash \texttt{downcastTT}(tt, c)$.*

To see how `downcastTT` works, observe how `(SUV) car` would be checked (in List. 6). To compute its typestate tree, we use `downcastTT`, defined in Def. 43, passing as parameter: (i) `(Car,ON,{})` as the typestate tree to downcast; (ii) `SUV` as the target class. Notice that, in the case the root is also a leaf, we need to replace it with the result of `downcastTT`. Concretely, $\texttt{downcastTT}((\texttt{Car}, \texttt{ON}, \{\}), \texttt{SUV}) = (\texttt{SUV}, \texttt{COMF_ON} \cup \texttt{SPORT_ON}, \{\})$.

4.3 Evolve

To compute the typestate tree of an object after a call, we define the `evolveTT` function.

▶ **Definition 45** (Evolve on typestate trees). *Function $\texttt{evolveTT} : \mathcal{TT} \times \mathcal{M} \times \mathcal{O} \rightarrow \mathcal{TT}$ is such that $\texttt{evolveTT}(tt, m, o)$ yields a new typestate tree resulting from applying $\texttt{evolve}(t, m, o)$ (Def. 31) to all the nodes of tt. The domain of `evolveTT` only includes triples (tt, m, o) such that $o \in \texttt{Ret}_m$ (i.e. the set of outputs returnable by method m). Formally,*

$$\texttt{evolveTT}((c, t, tts), m, o) = (c, \texttt{evolve}(t, m, o), \bigcup_{tt_i \in tts} \texttt{evolveTT}(tt_i, m, o)).$$

Notice that, when the set tts is empty, $\texttt{evolveTT}((c, t, tts), m, o) = (c, \texttt{evolve}(t, m, o), \{\})$

▶ **Theorem 46** (Typestate Trees Well-formedness Preserved By Evolve). *For all tt, m, o, such that $\vdash tt$, it holds that $\vdash \texttt{evolveTT}(tt, m, o)$.*

Listing 8 `EvolveTT` example.

```
1 Car c = new SUV();
2 if (c.turnOn()) c.turnOff();
```

To see how `evolveTT` works, consider the code presented in List. 8 (where the protocols of `Car` and `SUV` are presented in List. 1 and List. 2). The typestate tree of `c` is $(\texttt{Car}, \texttt{OFF}, \{(\texttt{SUV}, \texttt{OFF}, \{\})\})$. When the `turnOn` call occurs, we need to "evolve" each node of the typestate tree. To compute the resulting tree, we use `evolveTT`, defined in Def. 45, passing as parameter: (i) the typestate tree of `c`; (ii) `turnOn` as the method called; (iii) `true` as the expected output to enter the `if` branch. Concretely,

$$\texttt{evolveTT}((\texttt{Car}, \texttt{OFF}, \{(\texttt{SUV}, \texttt{OFF}, \{\})\}), \texttt{turnOn}, \texttt{true}) = (\texttt{Car}, \texttt{ON}, \{(\texttt{SUV}, \texttt{ON}, \{\})\}) .$$

Notice that every node of the typestate tree is "evolved" using the `evolve` function.

4.4 Merge

In the case of branching code, one has to merge type information coming from all different branches, so that subsequent code can be properly analysed by considering all possibilities (e.g., merging type information coming from both branches of an `if` statement). To this end, we define the `mrgTT` function, which merges two typestate trees. Before presenting `mrgTT`, we define some auxiliary functions, crucial for the formalisation.

▶ **Definition 47.** *Function $\texttt{height} : \mathcal{P}(\mathcal{TT}) \rightarrow \mathcal{N}$ is such that $\texttt{height}(tt)$ yields the greatest number of nodes traversed, in tt, from the root to one of the leaves (both included).*

▶ **Definition 48.** *Function $\texttt{clss} : \mathcal{P}(\mathcal{TT}) \rightarrow \mathcal{P}(\mathcal{C})$ is such that $\texttt{clss}(tts)$ yields the set of classes associated with the typestate trees in tts. Formally, $\texttt{clss}(tts) = \{\texttt{cl}(tt) \mid tt \in tts\}$.*

▶ **Definition 49.** *Function* $\texttt{find} : \mathcal{C} \times \mathcal{P}(\mathcal{TT}) \to \mathcal{TT}$ *is such that, given a class c and set of typestate trees tts with* $c \in \texttt{clss}(tts)$ *and* $\texttt{nodup}(tts)$, $\texttt{find}(c, tts)$ *yields the unique typestate tree in set tts whose class is c.*

▶ **Definition 50** (Merge)**.** *Function* $\texttt{mrgTT} : \mathcal{TT} \times \mathcal{TT} \to \mathcal{TT}$ *is such that, given typestate trees tt and tt',* $\texttt{mrgTT}(tt, tt')$ *yields the typestate tree obtained by merging tt and tt'. The domain of* $\texttt{mrgTT}$ *only includes pairs (tt, tt') such that* $\texttt{cl}(tt) = \texttt{cl}(tt')$*. Formally,*

$$\texttt{mrgTT}((c, t, tts), (c, t', tts')) = (c, t \cup t', tts_1 \cup tts_2 \cup tts_3)$$

$$\textit{where} \quad tts_1 = \bigcup_{c_i \in \texttt{clss}(tts) \cap \texttt{clss}(tts')} \texttt{mrgTT}(\texttt{find}(c_i, tts), \texttt{find}(c_i, tts'))$$

$$tts_2 = \bigcup_{c_i \in \texttt{clss}(tts) \setminus \texttt{clss}(tts')} \texttt{mrgTT}(\texttt{find}(c_i, tts), (c_i, \texttt{downcast}(t', c, c_i), \{\}))$$

$$tts_3 = \bigcup_{c_i \in \texttt{clss}(tts') \setminus \texttt{clss}(tts)} \texttt{mrgTT}((c_i, \texttt{downcast}(t, c, c_i), \{\}), \texttt{find}(c, tts'))$$

Note that `mrgTT` terminates since $\texttt{height}(tt) + \texttt{height}(tt')$ decreases with each recursive step. Moreover, `mrgTT` is symmetric, i.e., $\texttt{mrgTT}(tt, tt')$ gives the same result as $\texttt{mrgTT}(tt', tt)$.

▶ **Theorem 51** (Typestate Trees Well-formedness Preserved By Merge)**.** *For all tt, tt', such that* $\texttt{cl}(tt) = \texttt{cl}(tt')$*,* $\vdash tt$*, and* $\vdash tt'$*, it holds that* $\vdash \texttt{mrgTT}(tt, tt')$*.*

To see how `mrgTT` works, consider the `if` statement in List. 6 (lines 12-14). Notice that, although the else-branch is missing, in the process of computing the typestate tree of `car`, we need to consider it to be there (to account for all possible outputs returned by `switchMode`). To compute such typestate tree, we use `mrgTT`, defined in Def. 50, passing as parameters: (i) $(\texttt{SUV}, \texttt{SPORT_ON}, \{\})$ and (ii) $(\texttt{SUV}, \texttt{COMF_ON}, \{\})$. Since neither of the parameters have children nodes, it is enough to make the union of the root types. Concretely,

$$\texttt{mrgTT}((\texttt{SUV}, \texttt{SPORT_ON}, \{\}), (\texttt{SUV}, \texttt{COMF_ON}, \{\})) = (\texttt{SUV}, \texttt{SPORT_ON} \cup \texttt{COMF_ON}, \{\})\ .$$

5 Typestate Trees Soundness

In this section, we discuss why we consider type-safe a programming language equipped with our subtypestate mechanism. Such result relies on the key property that given a typestate tree that soundly approximates the current runtime typestate of an object, operations on it result in new typestate trees that still soundly approximate the runtime typestate. This assumes that class downcasts are performed to a class of which the object is a subtype of. So, we do not provide static guarantees that class downcasts will not throw at runtime.

▶ **Definition 52** (Sequence of upcasts on types)**.** *Function* $\texttt{upcast}^* : \mathcal{T} \times \mathcal{C} \times \mathcal{C} \to \mathcal{T}$ *is such that* $\texttt{upcast}^*(t, c, c')$ *performs zero or more upcasts from c to c' step-by-step, following the class hierarchy. The domain of* $\texttt{upcast}^*$ *only includes triples (t, c, c') such that* $\texttt{typestates}(t) \subseteq \texttt{ProtInputs}(c)$ *and* $c \leq_C c'$*. Formally,*

$$\texttt{upcast}^*(t, c, c') = \begin{cases} t & \textit{if } c = c' \\ \texttt{upcast}^*(\texttt{upcast}(t, c, \texttt{Super}(c)), \texttt{Super}(c), c') & \textit{otherwise} \end{cases}$$

Since the distance between c and c' decreases with each recursive step, $\texttt{upcast}^*$ terminates.

The next relation describes a typestate tree where type information is sound with respect to class c' and type t'. That is, assuming c' and t' represent the exact runtime type of a given object, a sound typestate tree correctly approximates such type. Note that the root has to be necessarily sound with respect to runtime, while the other nodes only need to be sound if the initialising class of the object is a subclass of the class associated with that node. Thus, all non-root nodes describe the type of the object if indeed the object is an instance of the corresponding class. This implies that if we downcast, possibly turning a non-root node into the new root, we preserve soundness only if the runtime downcast succeeds.

▶ **Definition 53** (Soundness Of Typestate Trees). *The predicate $\vdash_{c',t'}$ over $\mathcal{TT}$, with $\texttt{typestates}(t') \subseteq \texttt{ProtInputs}(c')$, asserts that, given a (well-formed) typestate tree (c, t, tts), it is sound with respect to class c' and type t'. Formally,*

$$\frac{c' \leq_C c \qquad \texttt{upcast}^*(t', c', c) \leq t \qquad \forall tt \in tts\ .\ c' \leq_C \texttt{cl}(tt)\ \Rightarrow \vdash_{c',t'} tt}{\vdash_{c',t'} (c, t, tts)}$$

The next theorem shows that soundness is preserved by typestate tree operations. Note that soundness after downcast is only preserved if at runtime the downcast does not throw an exception (thus the assumption $c \leq_C c' \leq_C \texttt{cl}(tt)$ on the second item of Theorem 54).

▶ **Theorem 54** (Typestate Trees Soundness Preservation). *Soundness is preserved by:*

upcast – *for all c, t, c', tt, such that $\vdash_{c,t} tt$ and $\texttt{cl}(tt) \leq_C c'$, it holds that* $\vdash_{c,t} \texttt{upcastTT}(tt, c')$

downcast – *for all c, t, c', tt, such that $\vdash_{c,t} tt$ and $c \leq_C c' \leq_C \texttt{cl}(tt)$, it holds that* $\vdash_{c,t} \texttt{downcastTT}(tt, c')$

evolve – *for all c, t, tt, m, o, such that $\vdash_{c,t} tt$, it holds that* $\vdash_{c,evolve(t,m,o)} \texttt{evolveTT}(tt, m, o)$

merge – *for all c, t, tt_1, tt_2, such that $\vdash_{c,t} tt_1$ or $\vdash_{c,t} tt_2$, and $\texttt{cl}(tt_1) = \texttt{cl}(tt_2)$, it holds that $\vdash_{c,t} \texttt{mrgTT}(tt_1, tt_2)$.*

Having shown our approach sound, the following section explains how the functions defined in Section 4 are used during type checking.

6 Application to type checking

We believe the setting presented is quite general and applicable to many object-oriented languages. In this section, we explain, how in detail, assuming a common syntax.[6] We start by describing how declarations are analysed in JaTyC, followed by expressions, and then statements. We use the Kleene star to denote (possibly empty) sequences.

Class declarations and overriding. First, it is crucial to guarantee that protocols are well-formed and the relation between classes and their protocols makes sense. To this, we ensure that all methods mentioned in the protocol are declared in the class. Similarly, we check that all mentioned outputs are return values of the corresponding methods. Additionally, we check typestate input contravariance and output covariance in overridden methods, since these may include `@Requires` and `@Ensures` annotations in parameters and return types, respectively, which limit the typestates received/returned. Finally, we ensure that the subclass protocol is

[6] Formalising a type checking system for one particular language, mechanising, and proving it sound is a matter for another paper. Doing that for a core Java-like language is work-in-progress.

a subtype of the superclass one (i.e., the initial state of the former is a subtype of the initial state of the latter according to Def. 4). Thanks to these checks, dynamic dispatch works transparently: if a method is callable on a supertype, it is also callable on its subtypes.

To type check a class, we analyse each method following the sequences of calls allowed by the protocol, similarly to the approach by Bravetti et al. [11], so that method analysis benefits from type information coming from the analyses of methods called before. Type information is stored in a map from locations (local variables, fields of the `this` object, and code expressions) to typestate trees (Def. 36). We also store the typestate trees of expressions since these may evaluate to typestated-objects, which must be tracked. Moreover, type information of final states is checked to ensure all fields either correspond to a terminated protocol or are aliased (explained later), to ensure *protocol completion* of references in fields.

Method declarations: `@Ensures(s) type m((@Requires(s) type x)*) {st}`. To check a method, we build a control flow graph [1] with the Checker Framework [30]. Then, we traverse it, visiting each expression or statement, and propagate type information. For each expression, we take the type information obtained from analysing the previous one, and produce new information. Expression or statement analysis is described later.

The initial type information (i.e., the initial input of the graph traversal) is composed by the types of the parameters, expressed via the `@Requires` annotation, combined with information about fields (coming from previous method analysis, as explained before). If a `@Requires` annotation is omitted, it means we expect an aliased reference. Return statements are analysed like assignments, while making sure the returned expression type is a subtype of the one declared via the `@Ensures` annotation. If no annotation is provided, we return an aliased reference. At the end of a method body, we ensure variables and code expressions are either aliased or in a final state, guaranteeing *protocol completion*.

Variable declarations or assignments: `[type] x = exp`. To check a variable declaration or assignment, we call `upcastTT` (Def. 39) on the typestate tree of the right-hand-side, and associate the result with the variable (or field) in the left-hand-side. If `upcastTT` yields a typestate tree with $\top$ as root type, the assignment is not allowed and we report an error. Given how the control flow graph is built, the expression on the right-side was already checked when we reach this point. If we override a typestate tree corresponding to a non-terminated protocol, we also report an error, since the assignment may compromise protocol completion of the overridden reference. Assignments may produce aliasing among variables. Since an object's state could be modified via multiple aliases, we restrict aliasing to allow us to statically track object states. We enforce a linear discipline: only one variable is "active", while the others are marked as aliased (and cannot call protocol methods). We also mark the right-hand-side expression as aliased when checking a variable declaration or assignment.

Method call expressions: `exp.m(exp*)`. To check a call, we first ensure the receiver expression is not `null`. We can do this because we distinguish between nullable and non-null types. Then, we analyse each *parameter assignment* applying the same rules explained before. This ensures that calls like `obj.m(x,x)` do not create unintended aliases. We also ensure that the root types of the typestate trees associated with the parameter expressions are subtypes of the expected types in the method signature. Following this, we proceed to check the call itself. We ensure the receiver expression is a non-aliased reference and use `evolveTT` (Def. 45) to compute the typestate tree associated with the receiver after the call, passing the current typestate tree, the method name, and a possibly returned output (if the method call appears in an `if` or `switch` statement). Note that `evolveTT` might be called several times to consider all possible outputs. If `evolveTT` yields a typestate tree where the root type is $\top$, then the method is not available to be called in that state, so we report an error.

Cast expressions: `(C) exp`. When checking a cast, we know that the inner expression was already checked, similarly to what happens to other expressions. To check it, we must use either `upcastTT` (Def. 39) or `downcastTT` (Def. 43), passing the inner expression typestate tree and the target class. We test if we are upcasting or downcasting by comparing the inner expression static type with the target class. The result is associated with the cast expression and the inner one is marked as aliased. As for assignments, if `upcastTT` yields a typestate tree with $\top$ in the root, we report an error. However, `downcastTT` does not produce errors because we provide type safety up-to downcasts not throwing runtime exceptions.

One key detail about cast expressions is that if a cast expression is the receiver object of a method call, after checking the call, the new type of the receiver object is associated with the most inner expression which is not a cast, not with the cast expression itself. For example, if the receiver is `(A) ((B) x)`, the new type information is associated with `x` directly, not with `(A) ((B) x)`, so that `x` can be used again later (instead of being aliased). This will require an upcast to the class of `x`, but no information is lost, thanks to typestate trees.

New expression: `new C(exp*)`. The initialisation of a new object is analysed similarly to a method call (since we are calling the constructor), except that it returns a new object. So, we associate the expression with a typestate tree with only a root: the class is the object type we are constructing, and the type (from Def. 10) is the initial typestate of the protocol.

If statements: `if ( exp ) { st } else { st' }`. For simplicity, up until now we omitted an implementation detail crucial to type check `if` statements (and `switch` statements): during the control flow graph traversal, we do not simply propagate a map from locations to typestate trees; we keep track of type information depending on the values of other expressions. For instance, to analyse a method call in a condition of an `if`, we track the type information for when the condition is `true` separately from the one when it is `false`. So, when checking an `if`, we just propagate the former to the first branch, and the latter to the second branch. We also make sure to invalidate such "conditional" type information once it is no longer relevant. Finally, the typestate trees associated with each location after the `if` are the result of merging type information from both branches, using `mrgTT` (Def. 50).

Switch statements: `switch ( exp ) { (case val : st)* }`. We analyse a `switch` statement similarly to an `if` one. A method call in the expression of a `switch` statement produces type information different for each `case`, but we consider enumeration values that may be returned instead of boolean values. So, to check a `switch` statement, we just need to propagate the information that holds when a given `case` is matched to the related branch. Again, we invalidate this "conditional" type information once it is no longer relevant.

While statements: `while ( exp ) { st }`. While statements are analysed like `if` ones, except the flow graph is different: after the body is executed, execution returns to the condition. Because of this, we might traverse the same expression or statement in the graph more than once. If that occurs, we merge the new gathered information with the previous one. To ensure that the static analysis terminates, we avoid analysing an expression again if no new type information was gathered. This is guaranteed to occur because the number of all possible typestates is finite. In the worst case, when merging, we might produce a union of all typestates. Typestate trees are also finite because the number of classes is finite.

7 Use Cases

To showcase the applicability and expressiveness of our approach, we start by explaining how the code in List. 6 is type checked in detail. Then, we present a suite of examples with polymorphic code[7], inspired from **cyber-physical systems**, showing that: (*i*) JaTyC detects errors the standard Java type checker does not detect; (*ii*) our setting is flexible and expressive enough to model interesting and intricate scenarios.

Type checking List. 6. To type check the `ClientCode` class (which has no protocol), we analyse the static methods `example` and `setSpeed`, independently (since static methods are not part of a class protocol). The list of steps to type check the `example` method follows:

- Check the expression `new SUV()`, associating it with a leaf typestate tree with class `SUV` and type `OFF` (i.e., $(\texttt{SUV}, \texttt{OFF}, \{\})$);
- Check the assignment, associating the previous typestate tree with the variable `suv`, and marking the expression on the right as aliased;
- Check the call `suv.turnOn()`, allowed in type `OFF`, generating “conditional” type information: if `true`, `suv` has typestate tree $(\texttt{SUV}, \texttt{COMF_ON}, \{\})$, otherwise it has $(\texttt{SUV}, \texttt{OFF}, \{\})$;
- Check the negating expression which “inverts” the conditional information;
- Inside the body of the `while` statement, `suv` is associated with $(\texttt{SUV}, \texttt{OFF}, \{\})$, and after exiting the `while`, `suv` is associated with $(\texttt{SUV}, \texttt{COMF_ON}, \{\})$;
- Check the call `suv.switchMode()`, which is allowed in type `COMF_ON`, generating “conditional” type information: `suv` has the typestate tree $(\texttt{SUV}, \texttt{SPORT_ON}, \{\})$ if the call returns `Mode.SPORT`, and if the call returns `Mode.COMFORT`, it has $(\texttt{SUV}, \texttt{COMF_ON}, \{\})$. Since the returned value is not checked in a `switch` statement, we combine both typestate trees into $(\texttt{SUV}, \texttt{SPORT_ON} \cup \texttt{COMF_ON}, \{\})$;
- Check the *parameter assignment* of `suv` by upcasting from `SUV` to `Car`, generating the typestate tree $(\texttt{Car}, \texttt{ON}, \{(\texttt{SUV}, \texttt{SPORT_ON} \cup \texttt{COMF_ON}, \{\})\})$. Since the root type `ON` is a subtype of the required type in the `@Requires` annotation, the *parameter assignment* is allowed. Additionally, variable `suv` is marked as aliased: the `setSpeed` method is now the one responsible to complete the protocol of the given instance;
- No further checks are necessary for the call expression on `setSpeed` since it is a static method and methods are checked in a modular way;
- Type checking the `example` method finishes by checking protocol completion. Since all locations are marked as aliased at the end, no error about completion is reported.

To finish checking the class, we analyse `setSpeed`. The list of steps taken follows:

- We associate `car` with typestate tree $(\texttt{Car}, \texttt{ON}, \{\})$, according to the `@Requires` annotation;
- Downcast from `Car` to `SUV`, resulting in the typestate tree $(\texttt{SUV}, \texttt{COMF_ON} \cup \texttt{SPORT_ON}, \{\})$;
- Check the call `((SUV)car).switchMode()`, which is allowed in type $\texttt{COMF_ON} \cup \texttt{SPORT_ON}$, generating “conditional” type information: `(SUV)car` has typestate tree $(\texttt{SUV}, \texttt{SPORT_ON}, \{\})$, if the call returns `Mode.SPORT`, and if the call returns `Mode.COMFORT`, it has $(\texttt{SUV}, \texttt{COMF_ON}, \{\})$;
- To make `car` usable again, upcast to `Car`, associating `car` with the typestate tree $(\texttt{Car}, \texttt{ON}, (\texttt{SUV}, \texttt{SPORT_ON}, \{\}))$, if the call returned `Mode.SPORT`; and $(\texttt{Car}, \texttt{ON}, (\texttt{SUV}, \texttt{COMF_ON}, \{\}))$ if the call returned `Mode.COMFORT`;

[7] The repository of our tool includes an `examples` folder containing such examples.

- Check the `if` statement by propagating the type information corresponding to each branch;
- In the body of the `if` statement, downcast (again) from `Car` to `SUV`, resulting in the typestate tree (SUV, SPORT_ON, {});
- Check the call `((SUV)car).setFourWheels(true)`, which is allowed in type SPORT_ON, in a similar fashion as before, associating `car` with (Car, ON, (SUV, SPORT_ON, {}));
- Merge type information from both branches, resulting in `car` being associated with typestate tree (Car, ON, (SUV, SPORT_ON $\cup$ COMF_ON, {}));
- Check the call `car.setSpeed(50)`, which is allowed in type ON, leading to ON;
- Check the call `car.turnOff()`, which is allowed in type ON, leading to OFF;
- Finish by checking protocol completion. Since all locations are marked as aliased or are in a final state (`car` is in the droppable typestate OFF), no completion error is reported.

Examples suite. We report the most significant examples of our suite in Table 1: **Directory** indicates the sub-directory; **Features** highlights the key features; **Checks** says if the example is accepted by our tool or not; and, **Runtime** describes the runtime error exhibited, if any.

In *Iterator (1)*, *Alarms* and *Cars (1)*, the examples test how our approach behaves with polymorphic code: as expected, the code compiles and no errors are thrown.

In *Drones (1)* and *Robots (1)*, the examples are more complex: we introduce a typestated data structure to increase the degree of flexibility (storing an arbitrary number of typestated objects, i.e., Drones and Robots). A key feature showcased here is the interaction between typestated objects: every time an object is used, it needs to be extracted from the data structure and put back once it has finished its task. In *Drones (2)*, the interaction between the data structure and the objects is even more articulate: we do not wait for the object to finish its task, but we immediately put it in the data structure and move to the next one, simulating a parallel tasks execution. The *Drones (3)* example is similar to the previous one, but it relies in a test for `null` being incorrectly negated in the return expression of an instance method, which causes a null pointer error in subsequent calls. The tool correctly propagates type information in the order methods may be called and catches this problem.

In *Iterator (2)* and *Cars (2)*, we show two problematic scenarios: index out of bounds and null-pointer exceptions, respectively. The former is caused by getting the next element without checking whether there are remaining elements or not. The latter is caused due to a field usage before initialising it. We are able to statically catch both cases.

Finally, in *Robots (2)*, we have another example of null-pointer error. The exception now is caused by a field being assigned to `null` in the subclass and used in the superclass, after performing an upcast. Thanks to our work, we are able to detect that, after assigning the field to `null`, the object is in a typestate with no supertypes, thus we raise an error.

In short, the provably sound theory presented in this paper is expressive and applicable to a mainstream object-oriented language, dealing with realistic code.

8 Related work

Fugue [13] allows checking typestates (seen as predicates over fields) by annotating methods with contracts and checking invariants. It handles casting and subclassing, where subclasses are allowed to introduce additional states with respect to superclasses. If an object ends up in a state unknown to its supertype, Fugue prohibits upcasting - as in our approach. To handle inheritance, *frame typestates* are introduced. Each frame is a set of fields declared in a particular class. An *object typestate* is the collection of frames. In our approach, our protocols

Table 1 Summary of examples.

Name	Directory	Features	Checks	Runtime
Iterator (1)	removable-iterator	Polymorphic safe code	Y	Ok
Iterator (2)	removable-iterator2	Wrong method call order	N	Out Of Bounds
Alarms	alarm-example	Polymorphic safe code	Y	Ok
Cars (1)	car-example	Polymorphic safe code	Y	Ok
Cars (2)	car-example2	Wrong method call order	N	Null-pointer
Drones (1)	drone-example	Typestated data structure	Y	Ok
Drones (2)	drone-example2	Typestated data structure Complex objects interaction	Y	Ok
Drones (3)	drone-example3	Same as Drones (2) and Incorrect test for `null`	N	Null-pointer
Robots (1)	robot-example	Typestated data structure Simple objects interaction	Y	Ok
Robots (2)	robot-example2	Wrong typestate upcast	N	Null-pointer

are globally defined with automata (e.g., List. 1 and 2), instead of method contracts, which we believe is more natural. Moreover, instead of using frames, we treat each class as a whole. This is enough since we view typestates as defining sequences of calls, not as predicates over fields, which simplifies the approach when dealing with overriding and dynamic dispatch.

Plural [8] statically checks that clients follow usage protocols based on typestates. It is based on earlier work [7] addressing the problem of substitutability of subtypes, while guaranteeing *behavioural subtyping* in an object-oriented language. Subtyping is supported by the programmer explicitly specifying which states "refine" (i.e., are substates of) others in the superclass. In our approach, we do not need to explicitly define subtyping relations: we define protocols in terms of state machines and automatically find all subtyping pairs.

Obsidian [12] is a language for smart contracts with a type system to statically detect bugs. It uses typestates to check state changes and has a permissions system for safe aliasing. It supports parametric polymorphism, but not casting to preserve strong static guarantees.

Gay et al. [18] extend earlier work on session types for object-oriented languages by attaching a protocol in the form of a session type to a class definition, and presenting an unification of communication channels and their session types, distributed object-oriented programming, and a form of typestates supporting non-uniform objects. The formal language includes a subtyping relation on session types [17] but does not include class inheritance (subtyping is just for channel communication). This approach has two implementations: Papaya [23] and Mungo [24]. Papaya considers protocols as in Gay et al. [18], but uses Scala as the target language with the same limitation of not coping with inheritance. Mungo considers protocols along the lines of Gay et al. [18], but uses Java (as we do) as the target object-oriented language. Inheritance is not supported apart from classes without protocols.

Bravetti et al. [11] present a type system for a Java-like language, where objects are annotated with usages, typestate-like specifications stating the allowed sequences of method calls. The type-based analysis ensures protocol compliance and completion, and memory safety (no null pointer dereferencing). However, subtyping (hence casting) is not supported.

Bouma et al. [10] develop a tool called BGJ that takes a global type, modelling the behaviour of processes in a multiparty session typing setting [20], and automatically generates Java classes modelling the APIs of projected local types. The state is encoded with a `state` field and transitions are encoded with methods annotated with preconditions and post-

conditions. To verify the clients of these APIs, the programmer writes Java code annotated with logical formulas. All annotations are statically checked by VerCors [9]. In our approach, one does not need to spread annotations throughout the code to specify or use protocols, we simply associate them with classes and the type system ensures memory-safety, protocol compliance and completion (properties the developer would need to specify for each program).

Table 2 Comparison of related work.

Work	How protocols are defined	Casting approach
Fugue	Typestates are seen as predicates over fields and methods annotated with contracts	Handles casting with frame typestates
Plural	States defined as "refinements" of superclass states and methods annotated with contracts	Explicit specification of subtyping relations
Obsidian	States defined explicitly and methods annotated with contracts	Casting disallowed for strong safety guarantees
Papaya	Usage types (i.e., automata-like)	Not supported
Mungo	Usage types (i.e., automata-like)	Not supported
BGJ	Scribble notation [28] projected to local types implemented as Java classes with `state` fields	Not supported
JaTyC (ours)	Usage types (i.e., automata-like)	Fully supported

9 Conclusions and future work

We overcome one of the main obstacles to the adoption of typestates in static analysis of object-oriented programs – the inability of performing cast operations freely at any point of the protocol – by introducing a novel theory based on typestate trees. We equip the theory with a set of functions to manage the typestate tree abstraction, and we mechanise soundness in the Coq proof system. We argue that typestate trees can be applied in various program analysers for object-oriented languages with inheritance, being thus language agnostic, opening the door for acceptance of several programs and features that were rejected until now in this kind of language. To support this claim, we implement a type checker for Java and assess the expressiveness of our approach. The relevance of the theory and of its applications is showcased by typestate-checking realistic Java code of an automotive system with driving dynamics control that allows to customise the drive mode of SUVs.

As future work, we plan to formally establish the runtime soundness of typestate trees by devising a core object-calculus with inheritance, static typestate semantics, and dynamic operational semantics, and by mechanising a type safety result: well-typed programs at runtime comply objects' protocol with respect to both the order of method calls and its completion, and do not raise null-pointer exceptions. Additionally, we will study how these concepts can be adapted to a setting with multi-inheritance and generics.

References

1. Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970. doi:10.1145/390013.808479.
2. Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. doi:10.1561/2500000031.

3 Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara. A Java typestate checker supporting inheritance. *Science of Computer Programming*, 221:102844, 2022. doi:10.1016/j.scico.2022.102844.

4 Lorenzo Bacchiani, Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. A Session Subtyping Tool. In *Proc. of Coordination Models and Languages (COORDINATION)*, volume 12717 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2021. doi:10.1007/978-3-030-78142-2_6.

5 Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and Union Types: Syntax and Semantics. *Information and Computation*, 119:202–230, 1995.

6 Nels E Beckman, Duri Kim, and Jonathan Aldrich. An Empirical Study of Object Protocols in the Wild. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, pages 2–26. Springer, 2011. doi:10.1007/978-3-642-22655-7_2.

7 Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005*, pages 217–226. ACM, 2005. doi:10.1145/1081706.1081741.

8 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, pages 301–320. ACM, 2007. doi:10.1145/1297027.1297050.

9 Stefan Blom and Marieke Huisman. The VerCors Tool for Verification of Concurrent Programs. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 127–131. Springer, 2014. doi:10.1007/978-3-319-06410-9_9.

10 Jelle Bouma, Stijn de Gouw, and Sung-Shik Jongmans. Multiparty Session Typing in Java, Deductively. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 19–27. Springer, 2023. doi:10.1007/978-3-031-30820-8_3.

11 Mario Bravetti, Adrian Francalanza, Iaroslav Golovanov, Hans Hüttel, Mathias Jakobsen, Mikkel Kettunen, and António Ravara. Behavioural Types for Memory and Method Safety in a Core Object-Oriented Language. In *Asian Symposium on Programming Languages and Systems*, volume 12470 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2020. doi:10.1007/978-3-030-64437-6_6.

12 Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ACM Trans. Program. Lang. Syst.*, 42(3):14:1–14:82, 2020. doi:10.1145/3417516.

13 Robert DeLine and Manuel Fähndrich. Typestates for Objects. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004. doi:10.1007/978-3-540-24851-4_21.

14 Edsger W. Dijkstra. The humble programmer, 1972. ACM Turing Award acceptance speech. doi:10.1145/355604.361591.

15 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 36(4):12, 2014. doi:10.1145/2629609.

16 Simon J. Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In *Proc. of Programming Languages and Systems (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1999. doi:10.1007/3-540-49099-X_6.

17 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi-calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/s00236-005-0177-z.

18 Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 299–312. ACM, 2010. doi:10.1145/1706299.1706335.

19 Tony Hoare. Null References: The Billion Dollar Mistake, 2009. Presentation at QCon London. URL: https://tinyurl.com/eyipowm4.

20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Type. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.

21 Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

22 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

23 Mathias Jakobsen, Alice Ravier, and Ornela Dardha. Papaya: Global Typestate Analysis of Aliased Objects. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP'21)*, pages 19:1–19:13. ACM, 2021. doi:10.1145/3479394.3479414.

24 Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J Gay. Typechecking protocols with Mungo and StMungo. In *Proc. of Principles and Practice of Declarative Programming (PPDP)*, pages 146–159. ACM, 2016. doi:10.1145/2967973.2968595.

25 Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. Casting about in the dark: an empirical study of cast operations in Java programs. *Proc. ACM Program. Lang.*, 3(OOPSLA):158:1–158:31, 2019. doi:10.1145/3360584.

26 João Mota, Marco Giunti, and António Ravara. On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage (Experience Paper). In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 40:1–40:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ECOOP.2023.40.

27 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proc. ACM Program. Lang.*, 2(OOPSLA):112:1–112:29, 2018. doi:10.1145/3276482.

28 Rumyana Neykova and Nobuko Yoshida. Featherweight Scribble. In Michele Boreale, Flavio Corradini, Michele Loreti, and Rosario Pugliese, editors, *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*, volume 11665 of *Lecture Notes in Computer Science*, pages 236–259. Springer, 2019. doi:10.1007/978-3-030-21485-2_14.

29 Jens Palsberg and Pavlopoulou Chirstina. From Polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, 2001. doi:10.1017/S095679680100394X.

30 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for Java. In *Proc. of Software Testing and Analysis (ISSTA)*, pages 201–212. ACM, 2008. doi:10.1145/1390630.1390656.

31 R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986. doi:10.1109/TSE.1986.6312929.

32 Vasco T. Vasconcelos. Sessions, from Types to Programming Languages. *Bull. EATCS*, 103:53–73, 2011. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/136.

A Research Methodology

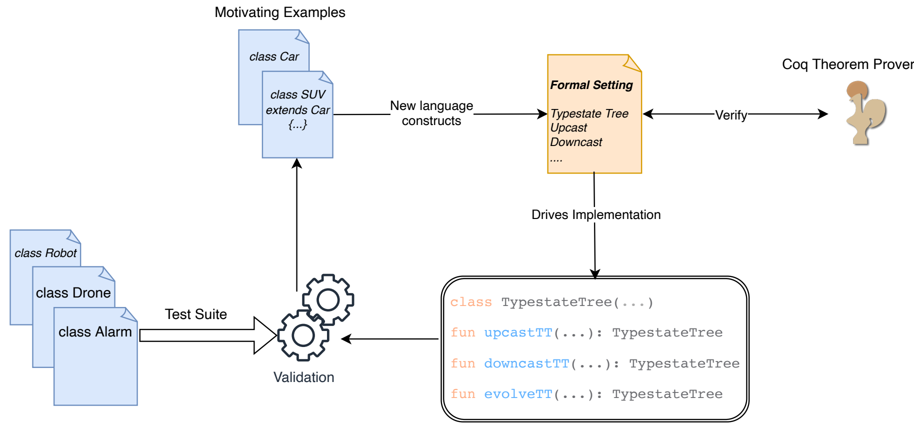


Figure 2 Research methodology behind the behavioural analysis support.

The iterative process in Figure 2 shows our methodology to support behavioural analysis within our type checker: we extract, from motivating examples, the language features to include in the static analysis; we build a formal setting (verified in the Coq theorem prover) to drive the JaTyC implementation; finally, we validate our approach with a suite of examples.

B Glossary

m A meta-variable ranging over the set of method identifiers **MNames**
o A meta-variable ranging over the set of output values **ONames**
s A meta-variable ranging over the set of typestate names **SNames**
$\widetilde{A}$ The wide tilde stands for a sequence of values
${}_d\{\widetilde{m : w}\}$ An input state (Definition 1)
$\langle\widetilde{o : u}\rangle$ An output state (Definition 1)
w A meta-variable ranging over input and output states, and typestate names (Definition 1)
u A meta-variable ranging over input states and typestate names (Definition 1)
$\widetilde{E}$ A set of defining equations
$w^{\widetilde{E}}$, $u^{\widetilde{E}}$ A meta-variable to denote a state w (resp. u) with a set of defining equations
$\mathcal{W}, \mathcal{U}$ The set of terms $w^{\widetilde{E}}$ (resp. $u^{\widetilde{E}}$)
$\mathcal{X}$ A subset of $w^{\widetilde{E}}$ containing only input states ${}_d\{\widetilde{m : w}\}$
$\mathcal{Y}$ A subset of $w^{\widetilde{E}}$ containing only output states $\langle\widetilde{o : u}\rangle$
$\leq_S$ A subtyping relation between states (Definition 4)
$\leq_{S_{alg}}$ The algorithmic version of the subtyping relation between states (Definition 7)
t A meta-variable ranging over the set of types $\mathcal{T}$ (Definition 10)
$\leq$ A subtyping relation between types (Definition 11)
c A meta-variable ranging over the set of class names $\mathcal{C}$
$\leq_C$ A subtyping relation between classes (Definition 15)
$\mathtt{Ret}_m$ The set of outputs returnable by a method m
tt A meta-variable ranging over the set of typestate trees $\mathcal{TT}$ (Definition 36)
tts A meta-variable ranging over $\mathcal{P}(\mathcal{TT})$
$\vdash$ Well-formedness of typestate trees (Definition 38)
$\vdash_{c,t}$ Soundness of typestate trees (Definition 53)