Dynamic Software Updates and Context Adaptation for Distributed Active Objects

Einar Broch Johnsen and Ingrid Chieh Yu

Abstract Dynamic software updates enable running programs to evolve without downtime. Drivers for such updates include software enhancements, bug fixes, and maintenance for software in domains, where it is costly, impractical, or even impossible to halt the system to reconfigure. In particular, application-level services in IoT ecosystems need to adapt seamlessly to changing, heterogeneous contexts. Services need to discover, adapt to, and interact with other services already deployed in the running ecosystem, supporting autonomicity within the service life-cycle. This paper explores a formalized, type safe asynchronous system of runtime software discovery and evolution, motivated by IoT ecosystems.

1 Introduction

Your life can be automated by connecting the objects around you. Having worn out your smart socks, you automatically received a pair of ultra-smart ones from the local retailer. You have just put them in your smart washing machine and wonder if its software is compliant with the latest generation of dirty socks. While you enjoy a drink from your intelligent water dispenser, your house adjusts the lights and rolls down the window blinds of your smart home, optimizing room temperature and humidity such that the smart laundry will dry as fast as possible, given your energy-friendly parameter settings. You live a smart life.

The Internet of Things (IoT) provides the underlying technologies for such a smart life, extending the Internet into the physical realm by connecting

Einar Broch Johnsen
Department of Informatics, University of Oslo, Norway, e-mail: einarj@ifi.uio.no

Ingrid Chieh Yu
Department of Informatics, University of Oslo, Norway, e-mail: ingridcy@ifi.uio.no

and combining spatially distributed devices with sensing and actuating capabilities. Application areas for IoT systems include smart homes, environmental monitoring, healthcare, and Industry 4.0. IoT devices are becoming commodity, and technologies from different providers need to interact as diverse devices are dynamically assembled into one system. Going beyond low-level interoperability, the IoT poses research challenges from a software engineering and systems perspective, including programmability, scalability, self-organization, security, and semantic interoperability. If IoT devices were to act as hosts for container-like services which can interact with the actuators, third party services can be deployed much like apps on a smart phone [22], maximizing the innovation potential of the IoT. This vision of IoT is naturally supported by programming with asynchronously communicating actors [1,15] or active objects [8], as evidenced by emerging technologies such as embedded actors [16], ELIoT [32], and Swarmlets [21].

Application-level services in IoT ecosystems need to adapt seamlessly to different contexts; we need methods to discover, deploy, and compose services at runtime, supporting autonomicity within the service life-cycle [26]. Techniques for software composition and service-oriented architectures offer a possible starting point to address these challenges based on interface encapsulation, such as design-by-contract [25], interface theories [23], and interfaces for service discovery and grouping [20]. However, a major challenge which lacks an established solution today, is the maintenance of services in this context. We can expect commodity IoT devices to need bugfixes and software enhancements after production. On the one hand, the continuous development of more advanced IoT services needs to combine with software already deployed on the devices. On the other hand, legacy devices make IoT services vulnerable to bugs and security issues. A recent study [33] reports that 45.5% of interviewed developers in the embedded software industry have no system for updating software on customers' devices beyond the physical replacement of devices; the other half have methods built in-house in the company. Although updating software on a deployed IoT device can technically be done via over-the-air firmware updates or through dynamic loading of binary modules [5], a challenge is to find high-level ways to program and orchestrate such updates at the abstraction level of the application logic.

Related Work. We briefly survey existing solutions for dynamic or online updates of deployed software which aim at modular evolution; some solutions keep multiple co-existing versions of a class or schema [4, 6, 7, 10, 11, 13, 17], others apply a global update or "hot-swapping" [2, 9, 24, 27]. The approaches differ for active behavior, which may be disallowed [9, 13, 24, 27], delayed [2], or supported [17, 34]. POLUS [10] supports life-cycle management of server software and the system-level workflow of dynamic updating. It is up to operators to control the process of dynamic updates such that one update will not interfere with another update and uses state synchronization functions to support multiple versions of data. At the language level, Hjálmtýsson and

Gray [17] propose proxy classes and reference indirection for C++, with multiple versions of each class. Old instances are not updated, so their activity is not interrupted. Existing approaches for Java, using proxies [27, 30] or modifying the Java virtual machine [24], use global update and do not apply to active objects. General-purpose dynamic software updating for single- and multithreaded C applications introduced in [14] updates the whole program instead of individual functions. The approach is tailored towards C developers, allowing programmer explicit control over the updating process by inserting update operations in long running loops (quiescent state), specific program points for when updates may take place as well as code to initiate data migration and redirection of execution to the new version. Automatic updates by lazy global update has been proposed for distributed objects [2] and persistent object stores [9], in which instances of updated classes are updated, but inheritance and (nonterminating) active code are not addressed, limiting the effect and modularity of the class update. In [9], the ordering of updates is serialized and in [24], invalid updates raise exceptions.

It is interesting to apply formal techniques to systems for software evolution. The engineering of dynamic system updates, addressed in [29], formalizes correctness criteria for updates, emphasizing safety in the sense that the resulting behavior after a dynamic update should be equivalent to an offline update. Interface automata have been used to characterize component-level updates [28] in terms of their interactions without considering implementation. For each kind, a state transformer indicates in which state and environment condition a component-based distributed system can be correctly updated. A runtime monitor is used to identify the matching rule and substitute the targeted component with the new version. Formalizations of programming-level runtime update mechanisms exist for imperative [34], functional [6], and object-oriented [7] languages. In a recent update system for (sequential) C [34], type-safe updates of type declarations and procedures may occur at annotated points identified by static analysis. However, the approach is synchronous as updates which cannot be applied immediately will fail. UpgradeJ [7] uses an incremental type system in which class versions are only typechecked once. UpgradeJ is synchronous and uses explicit update statements in programs. Updates only affect the class hierarchy and not running objects. Multiple versions of a class will coexist and the programmer must explicitly refer to the different class versions in the code.

Contribution of this Paper. This paper explores a system of software updates for the asynchronous evolution of distributed, loosely-coupled active objects, motivated by IoT systems. Our starting point is ABS [18], a formally defined active object language, in part building on Arnd Poetzsch-Heffter's work on JCoBox [31]. We revisit previous work on dynamic class updates [19, 36]; in this paper, the mechanism is simplified for ABS by focusing on distributed devices and interface encapsulation rather than on code reuse in class hierarchies, and combined with a simple interface query mechanism for context adaptation [20].

$$
\begin{array}{lll}
P & ::= \overline{D}\ \overline{L}\ \{\overline{T\ x}; sr\} & D ::= \textbf{interface}\ I\ \textbf{extends}\ \overline{I}\ \{\overline{M_s}\} \\
M_s & ::= T\ m\ (\overline{T\ x}) & L ::= \textbf{class}\ C(\overline{T\ x})\ \textbf{implements}\ \overline{I}\ \{\overline{T\ x}; \overline{M}\} \\
M & ::= M_s\{\overline{T\ x}; sr\} & s ::= x = e \mid \textbf{skip} \mid s; s \mid \textbf{if}\ c\ \{s\}\ \textbf{else}\ \{s\} \mid \textbf{suspend} \mid \textbf{await}\ g \\
sr & ::= s; \textbf{return}\ e & e ::= x \mid \textbf{new}\ C(\overline{e}) \mid e.\textbf{get} \mid e!m(\overline{e}) \mid \textbf{null} \\
b & ::= \mathsf{true} \mid \mathsf{false} \mid x & c ::= b \mid x\ \textbf{subtypeOf}\ I \\
T & ::= I \mid \mathsf{Bool} \mid \textbf{Fut}(T) & g ::= b \mid x? \mid g \wedge g
\end{array}
$$

Fig. 1 The language syntax. C is a class name, and I an interface name.

2 A Language for Distributed Active Objects

We define a small active object language with cooperative concurrency. The language syntax is given in Fig. 1. We emphasize the differences with Java. A program P consists of interface and class definitions, followed by a main block. An interface D has a name I, it extends a list $\overline{I}$ of inherited interfaces, and it contains a list $\overline{M_s}$ of method signatures. A class L has a name C, with parameters $\overline{x}$, implements a list of interfaces $\overline{I}$, defines a number of fields $\overline{x}$ typed by $\overline{T}$, and a list of methods $\overline{M}$. A method signature M_s associates a return type T and a list of formal parameters $\overline{x}$, typed by $\overline{T}$, to a method name m. A method definition M associates a method body to a method signature. The method body consists of local variables $\overline{x}$ of types $\overline{T}$ and list sr of statements to be executed; the list sr of statements ends with a final **return** statement. If a class contains a run method, this method is automatically activated on new instances of the class.

Statements s are standard apart from cooperative concurrency and interface querying. The statement **suspend** represents an unconditional suspension of the active process such that another suspended process may be scheduled. The statement **await** g represents a conditional suspension which depends on a guard g. If g evaluates to false, the statement gets preceded by a **suspend**, otherwise it becomes a **skip**. The conditional x **subtypeOf** I allows an object to be queried at runtime if it supports (a subtype of) a given interface. Such querying will be used for service discovery as software dynamically evolves.

Expressions e are standard apart from the (non-blocking) asynchronous method calls $e!m(\overline{e})$, which call method m on the object referenced by e with actual parameters referenced by $\overline{e}$, and the (blocking) operation x.**get** to access the value of the future referenced by x. *Guards* g are conjunctions of Boolean expressions b and polling operations $x?$ on futures x.

Example 1. Consider a smart home, where thermostat devices, with a temperature sensor and a heating actuator, are placed in different rooms and connected to a gateway. Each room has a specified target temperature (e.g., a bedroom is typically colder than the bathroom) and may contain several devices. The gateway will try to maintain a specified temperature for each room by adjusting the target temperature on the relevant devices. For convenience, the example uses a richer syntax than the calculus, including type synonyms

and basic types Int and Rat for integer and rational numbers. We also use standard operations over data types such as put and lookupDefault over maps (with constructor map[]) and appendright over lists (with constructor Nil and iterator **foreach**) from ABS [18]. Following ABS conventions, **return** statements are omitted for methods of return type Unit, variables can be declared where convenient, and write T x = **await** o!m() as syntactic sugar for the standard programming pattern T x; **Fut**(T) f; f = o!m(); **await** f?; t = f.**get**. The interface Any is the supertype of all interfaces.

We program the smart home by two classes SmartHomeGateway and Thermostat, shown in Fig. 2. In SmartHomeGateway, a method register is called by new devices to get connected. Note the use of interface querying to identify devices of type Thermostat, which are stored per location in a map, other devices are stored in the map unknown. The gateway monitors registered thermostats and receives their temperature values to, e.g., raise an alarm if the temperature exceeds a maximum temperature or collect statistics. The active method run calls itself asynchronously, allowing the cooperative scheduling of other processes between each monitoring cycle, so the gateway can register new sensors. The class Thermostat is connected to the IoT device through the TempSensor and HeatActivator interfaces. In Thermostat, the init block registers a new instance with the gateway. The Thermostat objects are active objects, i.e., their run method is automatically started to activate the heater, whenever the target temperature is not reached.

3 Programming Dynamic Software Updates

Software evolution may be perceived as a series of dynamic software updates injected into a running system, gradually modifying class definitions which interact the active objects. To enhance programmability, we consider operations U using the structuring concepts of the language, as given by the following syntax:

$$U ::= \textbf{newclass}\ C(\overline{T\ x})\ \textbf{implements}\ \overline{I}\ \{\overline{T\ x}; \overline{M}\} \mid \textbf{newinterface}\ I\ \textbf{extends}\ \overline{I}\ \{\overline{M_s}\}$$
$$\mid \textbf{update}\ C\ \textbf{implements}\ \overline{I}\ \{\overline{T\ x}; \overline{M}\}$$

The **newclass** declaration adds a new class C to the system, **newinterface** extends the type system with the new interface I, and **update** extends an existing class with new fields $\overline{x}$ and methods $\overline{M}$, *redefining* existing methods in the class in the case of name capture. Updates propagate *asynchronously* at runtime. They first modify classes, then the instances of those classes. We here ignore *state transfer* from an old to a new object state and initialize fields with default values; in a real system, an initialization block in the update could extend an old initialization block to give initial values to new fields in terms of the old fields.

```
module SmartHome;

type Loc = Int;
type Temp = Rat;

interface TempSensor { Temp read(); }
interface HeatActuator { Unit heat(); }
interface Thermostat { Temp sensing(); }
interface Gateway { Unit register(Loc loc, Thermostat t); }

def Map<T1,List<T2>> addItem<T1,T2>(Map<T1,List<T2>> store, T1 key, T2 item)
  = put(store,key, appendright(lookupDefault(store,key,Nil),item));

class SmartHomeGateway(Temp target) implements Gateway {
  Map <Loc, List<Thermostat>> thermostats = map[];
  Map <Loc, List<Any>> unknown = map[];
  Map <Loc, Temp> targetTemp = map[];
  Map <Thermostat, Temp> currentTemps = map[];
  Temp maxTemp = 50;

  Unit register(Loc loc, Any t){
    if (t subtypeOf Thermostat) thermostats=addItem(thermostats,loc, t);
    else unknown=addItem(unknown,loc, t); }

  Unit monitor(){
    List<Thermostat> mythermostats =
      foldl((List<Thermostat> v, List<Thermostat> a) =>
        concatenate(v, a))(values(thermostats),Nil);
    foreach (sensor in mythermostats){
      Temp tmp = sensor.sensing();
      if (tmp > maxTemp) ⟨...⟩; } } // Raise alarm...

  Unit updateTemperature(Thermostat t,Temp val){
    currentTemps=put(currentTemps,t,val);}

  Unit run(){ this.monitor(); this!run(); }
}

class Thermostat(Gateway gw, Temp targetTemp, Loc loc, TempSensor sensor,
  HeatActuator heater) implements Thermostat{

  { gw.register(loc, this); } // Init block:

  Rat sensing() { Temp t = await sensor!read(); return t; }

  Unit run(){
    Temp currentTemp = this.sensing();
    if (targetTemp > currentTemp) heater!heat(); // Activate heater
    this!run(); }
}
```

Fig. 2 The basic smart home example in ABS.

Example 2. We consider how to dynamically modify and add new services and devices into the smart home ecosystem of Example 1. Let us install smart window blinds in the smart home by introducing an active object to the interface WindowActuator. A class SmartWindow with a new interface WindowBlind allows each window blind at location loc to be connected to the gateway. The gateway can open and close blinds through method adjustBlind.

```
newinterface WindowBlind { Unit adjustBlind(); }
newinterface WindowActuator { Unit moveBlind(); }

newclass SmartWindow(ThermalComfort gw, Loc loc, WindowSensor sensor,
  WindowActuator controller) implements WindowBlind {

  { gw.register(loc, this); }

  Unit adjustBlind() { controller!moveBlind(); }
}
```

Example 3. Home owners may want to dynamically change the target temperature (e.g., 5 hours before coming home from a holiday). We update class Thermostat with a new method adjustTargetTemp which changes the target temperature of the thermostat device. The run method in class Thermostat will then use this new temperature setting in its next execution cycle.

```
newinterface Thermostat2 extends Thermostat { Unit adjustTargetTemp(Rat amount); }

update Thermostat implements Thermostat2{
  Unit adjustTargetTemp (Rat amount) { targetTemp = targetTemp + amount; }
}
```

Example 4. SmartHomeGateway needs to receive sensor values from the different thermostats and adjust the window blinds at different locations. The update operation in Fig. 3 shows modifications to class SmartHomeGateway. New mappings blinds and open will store the registered window blinds for each location and the state of the window blinds, and theromostats2 the enhanced thermostats. Method register is extended to recognize the new interfaces, and store devices accordingly.

The updated run method will monitor the old thermostats as before, but also monitor new thermostats and update currentTemps with the current sensor readings before calling adjustBlinds(loc) to adjust blinds at location loc. The new method adjustBlinds will open or close blinds for a given location, triggered by the temperature. The new method changeTargetTemp will allow home owners to change target room temperatures through the gateway.

By combining new interfaces, a new class WindowBlind, and updating the classes Thermostat and SmartHomeGateway, we obtain a very flexible dynamic update mechanism. However, modifications should not compromise the type safety of the running program. For example, class WindowBlind must be available before code to create objects of the class can run. Similarly, objects of class Thermostat must support adjustTargetTemp before the new SmartHomeGateway method changeTargetTemp can be called on these objects. In a distributed setting, where updates are not serialized but propagate asynchronously through the network, objects of different versions may coexist. Consequently, the order in which updates are applied at runtime may differ from the order in which they are type checked and inserted into the runtime system.

```
newinterface BlindController { Unit adjustBlind(); }
newinterface TempController { Unit changeTargetTemp(); }

update SmartHomeGateway implements BlindController, TempController {
  Map <Loc, List<Thermostat2>> thermostats2 = map[];
  Map <Loc,List<WindowBlind>> blinds = map[];
  Map <WindowBlind, Bool> open = map[];

  Unit register(Loc loc, Any t){
    if (t subtypeOf Thermostat2) thermostats2=addItem(thermostats2,loc, t);
    else { if (t subtypeOf Thermostat) thermostats=addItem(thermostats,loc, t);
           else if (t subtypeOf WindowBlind) blinds=addItem(blinds,loc, t);
                else unknown=addItem(unknown,loc, t); }}

  Unit adjustBlinds(Loc loc){
    List<Thermostat> sensors = lookupDefault(thermostats,loc,Nil);
    Temp targetLoc = lookupDefault(targetTemp,loc,target);
    List<WindowBlind> blinds = lookupDefault(blinds,loc,Nil);
    foreach (sensor in sensors){
      Temp current = sensor.sensing();
      currentTemps = put(currentTemps,sensor,current); }
    List<Temp> temps = map((Thermostat sensor)=>
                             lookupDefault(currentTemps,sensor,0))(sensors);
    Temp avg = average(temps);
    if (average(temps) <targetLoc)
      foreach (blind in blinds){
        if(lookupDefault(open,blind,False)){ blind.adjustBlind(); put(open,blind,True); }}
    else
      foreach (blind in blinds){
        if(lookupDefault(open,blind,True)){ blind.adjustBlind(); put(open,blind,False); }}}

  Unit changeTargetTemp(Loc loc, Int newTemp){
    List<Thermostat2> ts = lookupDefault(thermostats2,loc,Nil);
    targetTemp=put(targetTemp, loc, newTemp);
    foreach (t in ts){ t!adjustTargetTemp(newTemp); }}

  Unit newDevices(){
    foreach (loc in elements(keys(unknown))) {
      foreach (t in lookupDefault(unknown,loc,Nil)){
       if (t subtypeOf Thermostat2) thermostats2=addItem(thermostats2,loc,t);
       if (t subtypeOf WindowBlind) blinds=addItem(blinds,loc, t); }}}

  Unit run(){
    this.newDevices();
    this.monitor();
    foreach (loc in elements(keys(targetTemp))) {
      foreach (t in lookupDefault(thermostats,loc,Nil));{
        Temp newtemp = t.sensing(); updateTemperature(t,newtemp); adjustBlinds(loc); }
      foreach (t in lookupDefault(thermostats2,loc,Nil)){
        Temp newtemp = t.sensing(); updateTemperature(t,newtemp); adjustBlinds(loc); }}
    this!run();}
}
```

Fig. 3 Updating the gateway.

4 Type Checking for Asynchronous Software Updates

To statically track dependencies to different versions of code, we consider a type and effect system [3, 35] developed in the context of a *mapping family*.

Definition 1. Let n be a name, δ a declaration, $i \in \mathcal{I}$ a mapping index, and $[n{\mapsto_i}\delta]$ the binding of n to δ indexed by i. A *mapping family* Γ is built from the empty mapping family $\emptyset$ and indexed bindings by the constructor $+$. The *extraction* of an indexed mapping Γ_i from Γ and *application* for the indexed mapping Γ_i, are defined as follows

$$\begin{array}{ll} \emptyset_i & = \varepsilon \\ (\Gamma + [n{\mapsto_{i'}}\delta])_i & = \textbf{if } (i = i') \textbf{ then } \Gamma_i + [n{\mapsto_i}\delta] \textbf{ else } \Gamma_i \\ \\ \varepsilon(n) & = \bot \\ (\Gamma_i + [n{\mapsto_i}\delta])(n') & = \textbf{if } (n = n') \textbf{ then } \delta \textbf{ else } \Gamma_i(n'). \end{array}$$

Our typing context uses four indexes; the mappings $\Gamma_\mathcal{I}$ and $\Gamma_\mathcal{C}$ map interface and class names to interface and class declarations, and Γ_v maps program variable names to types. In a static setting, $\Gamma_\mathcal{I}$ and $\Gamma_\mathcal{C}$ correspond to the *static tables* which can usually be omitted from the type system, but this is not the case with dynamic software updates. For the purposes of dynamic updates, there is a mapping of *dependencies* $\Gamma_d : \mathsf{Dep} \to \mathsf{Set}[\mathsf{Dep}]$, where the type Dep consist of pairs of class names and natural numbers. A version of a class C can be uniquely identified by a natural number; e.g., $\langle C, 5\rangle$ represents the fifth update of C. Elements in $\Gamma_d(\langle C, u\rangle)$ will represent versions of classes on which an update u of a class C depends; these dependencies are inferred from the current class table by the type analysis, and exploited for dynamic classes in Sect. 5.

We assume for simplicity that variable declarations $\overline{T\ x}$ have unique names and known types, and denote by $[\overline{x}{\mapsto_v}\overline{T}]$ the mapping (built from the bindings $[x{\mapsto_v}T]$). The auxiliary function *implements*(C, I, Γ) matches signatures for methods declared in an interface I to those in C, in order to check that the class provides method bodies for the method declarations of its interfaces. We omit the (straightforward) definitions of other auxiliary functions on Γ; e.g., *curr*(C, Γ) denotes the current version of C in Γ.

Typing of the Base Language. The (static) base language uses a standard type system for object-oriented languages with explicit futures (e.g., [18]), decorated with effects. Subtyping $T_1 \preceq T_2$ is defined by interface inheritance: If I extends J then $J \preceq I$. The type rules are given in Fig. 4. Judgments have the form $\Gamma \vdash e : T\ \langle\Sigma\rangle$ and $\Gamma \vdash s\ \langle\Sigma\rangle$, where Γ is the typing environment and $\Sigma : \mathsf{Set}[\mathsf{Dep}]$ the effect. To simplify the presentation, we assume that method declarations in interfaces are unique and well-typed and omit the analysis of interfaces. In rule Conditional2, the interface query for x changes the type of x in s_1. Note that in New, the current version of the instantiated class is recorded in the effect (highlighted). These effects are collected for each method, and assembled to update the dependency mapping (highlighted) of the class in Class and again collected in Program (highlighted).

$$\textsc{(Suspend)}\ \Gamma \vdash \textbf{suspend} : \mathsf{ok} \qquad \textsc{(Skip)}\ \Gamma \vdash \textbf{skip} : \mathsf{ok} \qquad \textsc{(Null)}\ \Gamma \vdash \textbf{null} : I \qquad \textsc{(Var)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\textsc{(Poll)}\ \frac{\Gamma \vdash x : \textbf{Fut}(T)}{\Gamma \vdash x? : \mathsf{Bool}} \qquad \textsc{(Get)}\ \frac{\Gamma \vdash x : \textbf{Fut}(T)}{\Gamma \vdash x.\textbf{get} : T} \qquad \textsc{(Assign)}\ \frac{\Gamma \vdash e : \Gamma(x)\,\langle\Sigma\rangle}{\Gamma \vdash x := e : \mathsf{ok}\,\langle\Sigma\rangle} \qquad \textsc{(Await)}\ \frac{\Gamma \vdash g : \mathsf{Bool}}{\Gamma \vdash \textbf{await}\ g : \mathsf{ok}}$$

$$\textsc{(Call)}\ \frac{\Gamma \vdash \overline{e} : T\,\langle\Sigma_1\rangle \qquad \Gamma \vdash e : I\,\langle\Sigma_2\rangle \qquad match(m, T \to T', I)}{\Gamma \vdash e!m(\overline{e}) : \textbf{Fut}(T')\,\langle\Sigma_1 \cup \Sigma_2\rangle} \qquad \textsc{(Composition)}\ \frac{\Gamma \vdash s : \mathsf{ok}\,\langle\Sigma_1\rangle \qquad \Gamma \vdash s' : \mathsf{ok}\,\langle\Sigma_2\rangle}{\Gamma \vdash s; s' : \mathsf{ok}\,\langle\Sigma_1 \cup \Sigma_2\rangle} \qquad \textsc{(And)}\ \frac{\Gamma \vdash g_1 : \mathsf{Bool} \qquad \Gamma \vdash g_2 : \mathsf{Bool}}{\Gamma \vdash g_1 \wedge g_2 : \mathsf{Bool}}$$

$$\textsc{(New)}\ \frac{T \in interfaces(\Gamma_{\mathcal{C}}(C)) \qquad \Gamma \vdash \overline{e} : type(\Gamma_{\mathcal{C}}(C).param)}{\Gamma \vdash \textbf{new}\ C(\overline{e}) : T\ \langle (C, curr(C, \Gamma))\rangle} \qquad \textsc{(Conditional1)}\ \frac{\Gamma \vdash b : \mathsf{Bool} \qquad \Gamma \vdash s_1 : \mathsf{ok}\,\langle\Sigma_1\rangle \qquad \Gamma \vdash s_2 : \mathsf{ok}\,\langle\Sigma_2\rangle}{\Gamma \vdash \textbf{if}\ b\ \{s_1\}\ \textbf{else}\{s_2\} : \mathsf{ok}\,\langle\Sigma_1 \cup \Sigma_2\rangle}$$

$$\textsc{(Sub)}\ \frac{T \preceq T' \qquad \Gamma \vdash e : T'}{\Gamma \vdash e : T} \qquad \textsc{(Conditional2)}\ \frac{\Gamma[x \mapsto_v I] \vdash s_1 : \mathsf{ok}\,\langle\Sigma_1\rangle \qquad \Gamma \vdash s_2 : \mathsf{ok}\,\langle\Sigma_2\rangle}{\Gamma \vdash \textbf{if}\ x\ \textbf{subtypeOf}\ I\ \{s_1\}\ \textbf{else}\{s_2\} : \mathsf{ok}\,\langle\Sigma_1 \cup \Sigma_2\rangle}$$

$$\textsc{(Method)}\ \frac{\Gamma' = \Gamma + [\overline{x} \mapsto_v \overline{T}, \overline{x'} \mapsto_v \overline{T'}] \qquad \Gamma' \vdash e : T'\,\langle\Sigma_1\rangle \qquad \Gamma' \vdash s : \mathsf{ok}\,\langle\Sigma_2\rangle}{\Gamma \vdash T'\ m\ (\overline{T\ x})\{\overline{T'\ x'}; s; \textbf{return}\ e\} : \mathsf{ok}\,\langle\Sigma_1 \cup \Sigma_2\rangle}$$

$$\textsc{(Program)}\ \frac{\Gamma[\overline{x} \mapsto_v \overline{T}] \vdash s : \mathsf{ok} \qquad \forall L \in \overline{L} \cdot \Gamma + \Gamma_d^L \vdash L : \mathsf{ok}}{\Gamma + \bigcup_{L \in \overline{L}} \Gamma_d^L \vdash \overline{L}\ \{\overline{T\ x}; s\} : \mathsf{ok}}$$

$$\textsc{(Class)}\ \frac{\forall I \in \overline{I} \cdot implements(C, I, \Gamma) \qquad \forall M \in \overline{M}\ \cdot\ \Gamma[\textbf{this} \mapsto_v C, \overline{x} \mapsto_v \overline{T}, \overline{x'} \mapsto_v \overline{T'}] \vdash M : \mathsf{ok}\,\langle\Sigma^M\rangle}{\Gamma + [\langle C, 0\rangle \mapsto_d \bigcup_{M \in \overline{M}} \Sigma^M] \vdash \textbf{class}\ C(\overline{T\ x})\ \textbf{implements}\ \overline{I}\ \{\overline{T'\ x'}; \overline{M}\} : \mathsf{ok}}$$

Fig. 4 The type and effect system. Judgments for the Boolean constants true and false are similar to Suspend. We omit empty effects; e.g., $\Gamma \vdash e\,\langle\emptyset\rangle$ is written $\Gamma \vdash e$.

Typing of Dynamic Class Extensions. Dynamic class operations $U_1, U_2, \ldots$ are type checked in a *sequence* of typing environments $\Gamma^0, \Gamma^1, \ldots$, which extend each other; Γ^0 is the typing environment for the original program and Γ^i the current static view of the system. We describe the construction of Γ^{i+1} for the next well-typed update U. The type system for judgments $\Gamma^{i+1} \vdash U$ is shown in Fig. 5. We here omit the analysis of **newinterface** and focus on class updates, and assume that new interfaces are well-typed and that $\Gamma_{\mathcal{I}}^i$ is correctly extended for each update. (As before, we omit the straightforward analysis of superinterfaces and method signatures.)

Rule New-Class for class additions requires a fresh name, type checks like a class in the original program, and extends $\Gamma_{\mathcal{C}}^i$. The version number of the new class differs from those of the program's original classes (highlighted). This reflects that the new class may depend on other dynamic system changes.

$$\frac{\begin{array}{c}\text{(New-Class)}\\ \Gamma' = [C{\mapsto_C}(\overline{T\ x}, \overline{I}, \overline{T'\ x'}, \overline{M})] \quad \forall I \in \overline{I} \cdot \mathit{implements}(C, I, \Gamma + \Gamma') \\ C \notin \mathit{dom}(\Gamma^i_{\mathcal{C}}) \quad \forall M \in \overline{M} \ \cdot \ \Gamma^i + \Gamma'[\mathsf{this} \mapsto_v C, \overline{x} \mapsto_v \overline{T}, \overline{x'} \mapsto_v \overline{T'}] \vdash M : \mathsf{ok}\,\langle \Sigma^M \rangle\end{array}}{\Gamma^i + \Gamma' + \ [\langle C, 1\rangle {\mapsto_d} \bigcup_{M \in \overline{M}} \Sigma^M] \ \vdash \mathbf{newclass}\ C(\overline{T\ x})\ \mathbf{implements}\ \overline{I}\ \{\overline{T'\ x'}; \overline{M}\}}$$

$$\frac{\begin{array}{c}\text{(Class-Extend)}\\ \Gamma^i_{\mathcal{C}}(C) = (\overline{T'\ x'}, \overline{I}', \overline{T''\ x''}, \overline{M}') \quad \forall I \in \overline{I} \cdot \mathit{implements}(C, I, \Gamma^i + \Gamma') \\ \mathit{refines}(\overline{M}, \overline{M}') \qquad \Gamma' = [C \mapsto_C (\overline{T'\ x'}, \overline{I}'; \overline{I}, \overline{T''\ x''}; \overline{T\ x}, (\overline{M}' \oplus \overline{M}))] \\ \mathit{vs} = \mathit{curr}(C, \Gamma^i_d) \qquad \forall M \in \overline{M} \ \cdot \ \Gamma^i + \Gamma'[\mathsf{this} \mapsto_v C, \overline{x} \mapsto_v \overline{T}] \vdash M\,\langle \Sigma^M \rangle\end{array}}{\Gamma^i + \Gamma' + \ [\langle C, \mathit{vs}+1\rangle {\mapsto_d} \bigcup_{M \in \overline{M}} \Sigma^M \cup \{(C, \mathit{vs})\}] \ \vdash \mathbf{update}\ C\ \mathbf{implements}\ \overline{I}\ \{\overline{T\ x}; \overline{M}\}}$$

Fig. 5 The type system for dynamic class extensions. Judgments have the form $\Gamma^{i+1} \vdash U$, where Γ^i is the current typing environment before the operation U.

Rule Class-Extend obeys a substitutability discipline captured by the predicate $\mathit{refines}(\overline{M}, \overline{M}')$; if $M \in \overline{M}$ redefines $M' \in \overline{M}'$, the signature of M must be a subtype of the signature of M'. When retrieving the old version of the class from Γ^i_C, we represent the class compactly as a tuple and we denote by $\overline{M}' \oplus \overline{M}$ the union operation which retains methods in $\overline{M}$ in case of name conflicts. The new definition of C replaces the old one in $\Gamma^i_{\mathcal{C}}$ by the binding Γ'. The function $\mathit{curr}(C, \Gamma^i_d)$ identifies the current version number of a class C by inspecting the dependency mapping. The new methods of the updated class are type checked in a similar way as rule Class and the resulting dependencies, accumulated by the type analysis of methods, are bound to the new version $\mathit{curr}(C, \Gamma^i_d) + 1$ of the class in Γ^{i+1}_d (highlighted). Since type checking assumes the current version of the class to be present, the current version is included in this mapping.

In the asynchronous distributed setting, updates may be delayed or bypass each other. The *static view* of the system, reflected by the current typing environment, may differ considerably from the running system. To ensure that execution is type safe, inferred dependencies in Γ^i are imposed as constraints on the applicability of the i'th update at runtime. The constraints ensure that updates which depends on each other will be applied in the correct order; otherwise, they may be applied in any order, or in parallel.

5 Context-Reduction Semantics

The semantics is given by a small-step reduction relation $\mathit{rcf} \rightarrow \mathit{rcf}'$ over configurations. *Configurations* consist of objects, futures, and a class table CT (see Fig. 6). A *class* has a version number, parameters, a list of interfaces, a set of fields with default values, and a set of methods, bound to the class id in CT.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathit{rcf} & ::= & \mathit{CT} \rhd \mathit{cf} & \mathit{object} & ::= & o(n, a, \mathit{active}, q) \\
\mathit{CT} & ::= & C \mapsto \langle n, \overline{x}, \overline{I}, a, \mathit{mtds}\rangle \mid \mathit{CT} \circ \mathit{CT} & a & ::= & x \mapsto \langle T, v\rangle \mid a \circ a \\
\mathit{cf} & ::= & \epsilon \mid \mathit{object} \mid \mathit{future} \mid \mathit{msg} \mid \mathit{cf}\ \mathit{cf} & \mathit{active} & ::= & \mathit{pr} \mid \texttt{idle} \\
\mathit{future} & ::= & f \mid f(v) & q & ::= & \epsilon \mid \mathit{pr} \mid q \circ q \\
\mathit{msg} & ::= & \mathit{bind}(C, m, \overline{v}, o, f) \mid \mathit{bound}(o, \mathit{pr}) & \mathit{pr} & ::= & \{a|sr\} \mid \texttt{error} \\
\mathit{mtd} & ::= & T\ m(\overline{T}\ \overline{x})\{a|sr\} & v & ::= & o \mid f \mid \texttt{null} \mid b \\
\mathit{mtds} & ::= & \epsilon \mid \mathit{mtd} \mid \mathit{mtds}\ \mathit{mtds} & & &
\end{array}
$$

Fig. 6 Syntax for runtime configurations; o and f are object and future identifiers.

Default values for types are given by a function *default* (e.g., $\mathit{default}(\mathsf{bool}) = \mathsf{false}$, and $\mathit{default}(I) = \mathit{default}(\mathsf{Fut}(T)) = \mathbf{null}$). An *object* has an id o, version n, fields a, an active process, and a queue q of suspended processes. The `idle` process indicates that no method is active in the object and `error` that method binding has failed. A *future* has an id f, which becomes $f(v)$ when the future's reply value v has been received. Denote by $\overline{I}^*$ the transtive closure of $\overline{I}$ over the subtype relation (i.e., if $I \in \overline{I}$ and $J \preceq I$, then $I, J \in \overline{I}^*$). The *initial configuration* of a program $\overline{L}\ \{\overline{T\ x}; sr\}$, with classes and one *main* object, is defined as follows:

Definition 2 (Initial Configuration). Let P be a program and let $\mathit{Main} = \mathit{classOf}(\mathit{main})$. The initial configuration for P, denoted $[\![P]\!]$, is defined as:

$$
\begin{array}{ll}
[\![\overline{D}\ \overline{L}\ \{\overline{T}\ \overline{x}; s\}]\!] & = \mathit{Main} \mapsto \langle \mathbf{this} \mapsto \langle \mathit{Main}, \mathbf{null}\rangle, \epsilon\rangle \circ [\![\overline{L}]\!] \\
 & \quad \rhd \mathit{main}(0, \mathbf{this} \mapsto \langle \mathit{Main}, \mathbf{null}\rangle, \{[\![\overline{T}\ \overline{x}]\!]|s\}, \epsilon) \\
[\![L\ \overline{L}]\!] & = [\![L]\!] \circ [\![\overline{L}]\!] \\
[\![\mathbf{class}\ C(\overline{T\ x})\ \mathbf{implements}\ \overline{I}\ \{\overline{T'\ x'}; \overline{M}\}]\!] & \\
 & = C\ \mapsto \langle 0, \overline{x}, \overline{I}^*, \langle \mathbf{this} \mapsto \langle C, \mathbf{null}\rangle \circ [\![\overline{T\ x}, \overline{T'\ x'}]\!], [\![\overline{M}]\!]\rangle \\
[\![T\ x, \overline{T}\ \overline{x}]\!] & = [\![T\ x]\!] \circ [\![\overline{T}\ \overline{x}]\!] \\
[\![T\ x]\!] & = x \mapsto \langle T, \mathit{default}(T)\rangle \\
[\![M\ \overline{M}]\!] & = [\![M]\!]\ [\![\overline{M}]\!] \\
[\![T\ m(\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'}; sr\}]\!] & = T\ m(\overline{T}\ \overline{x})\{\mathsf{destiny} \mapsto \langle \mathsf{Fut}(T), \mathbf{null}\rangle \circ [\![\overline{T}\ \overline{x}, \overline{T'}\ \overline{x'}]\!]|sr\}
\end{array}
$$

We consider a context reduction semantics [12], which decomposes a statement into a reduction context and a redex, and reduces the redex. The main reduction rules are given in Fig. 7. *Reduction contexts* are method bodies M, statements S, expressions E, and guards G with a single hole denoted by $\bullet$:

$$
\begin{array}{lll}
M & ::= \bullet \mid S; \mathbf{return}\ e \mid \mathbf{return}\ E & S ::= \bullet \mid v = E \mid S; s \mid \mathbf{if}\ G\ \{s_1\}\ \mathbf{else}\ \{s_2\} \\
E & ::= \bullet \mid E.\mathbf{get} \mid E!m(\overline{e}) \mid o!m(\overline{v}, E, \overline{e}) & G ::= \bullet \mid E? \mid G \wedge g \mid b \wedge G \mid E\ \mathbf{subtypeOf}\ I
\end{array}
$$

Redexes reduce in their respective contexts; i.e., body-redexes in M, stat-redexes in S, expr-redexes in E, and guard-redexes in G. Redexes are defined as follows:

$$
\begin{array}{ll}
\mathit{body\text{-}redexes} & ::= \mathbf{skip} \mid \mathbf{return}\ v \\
\mathit{stat\text{-}redexes} & ::= x = v \mid \mathbf{await}\ g \mid \mathbf{skip}; s \mid \mathbf{if}\ b\ \{s\}\ \mathbf{else}\ \{s\} \mid \mathbf{suspend} \\
\mathit{expr\text{-}redexes} & ::= x \mid f.\mathbf{get} \mid o!m(\overline{v}) \mid \mathbf{new}\ C(\overline{v}) \\
\mathit{guard\text{-}redexes} & ::= f? \mid b \wedge g
\end{array}
$$

$$\frac{l^T(x) = T}{o(n,a,\{l|M[x=v]\},q) \to o(n,a,\{l[x \mapsto \langle T,v\rangle]|M[\textbf{skip}]\},q)}\ (\textsc{Red-Assign1})$$

$$\frac{x \notin dom(l) \quad a^T(x) = T}{o(n,a,\{l|M[x=v]\},q) \to o(n,a[x \mapsto \langle T,v\rangle],\{l|M[\textbf{skip}]\},q)}\ (\textsc{Red-Assign2})$$

$$o(n,a,\{l|M[\textbf{if true } \{s\} \textbf{ else } \{s'\}]\},q) \to o(n,a,\{l|M[s]\},q)\ (\textsc{Red-Cond1})$$

$$o(n,a,\{l|M[\textbf{if false } \{s\} \textbf{ else } \{s'\}]\},q) \to o(n,a,\{l|M[s']\},q)\ (\textsc{Red-Cond2})$$

$$o(n,a,\{l|M[\textbf{skip}; s]\},q) \to o(n,a,\{l|M[s]\},q)\ (\textsc{Red-Skip1})$$

$$o(n,a,\{l|\textbf{skip}\},q) \to o(n,a,\texttt{idle},q)\ (\textsc{Red-Skip2})$$

$$o(n,a,\{l|M[\textsf{true} \wedge G]\},q) \to o(n,a,\{l|M[G]\},q)\ (\textsc{Guard1})$$

$$o(n,a,\{l|M[\textsf{false} \wedge G]\},q) \to o(n,a,\{l|M[\textsf{false}]\},q)\ (\textsc{Guard2})$$

$$\frac{o' \text{ is fresh} \quad CT(C) = \langle n,\overline{x},\overline{I},a,mtds\rangle \quad C = classOf(o') \quad a'' = [\textbf{this} \mapsto \langle C,o'\rangle, \overline{x} \mapsto \langle a^T(\overline{x}),\overline{v}\rangle]}{CT \rhd o(n',a',\{l|M[\textbf{new } C(\overline{v})]\},q) \to CT \rhd o(n',a',\{l|M[o']\},q)\ o'(n,a;a'',\texttt{idle},\varepsilon)}\ (\textsc{Red-New})$$

$$\frac{C = classOf(v) \quad I \in \overline{I} \quad CT(C) = \langle n',\overline{x},\overline{I},a',mtds\rangle}{CT \rhd o(n,a,\{l|M[v \textbf{ subtypeOf } I]\},q)\ f(v) \to CT \rhd o(n,a,\{l|M[\textsf{true}]\},q)\ f(v)}\ (\textsc{Red-Query1})$$

$$\frac{C = classOf(v) \quad I \notin \overline{I} \quad CT(C) = \langle n',\overline{x},\overline{I},a',mtds\rangle}{CT \rhd o(n,a,\{l|M[v \textbf{ subtypeOf } I]\},q)\ f \to CT \rhd o(n,a,\{l|M[\textsf{false}]\},q)\ f}\ (\textsc{Red-Query2})$$

$$o(n,a,\{l|M[f?]\},q)\ f(v) \to o(n,a,\{l|M[\textsf{true}]\},q)\ f(v)\ (\textsc{Red-Poll1})$$

$$o(n,a,\{l|M[f?]\},q)\ f \to o(n,a,\{l|M[\textsf{false}]\},q)\ f\ (\textsc{Red-Poll2})$$

$$\frac{o' \neq \textbf{this} \quad C = classOf(o') \quad f \text{ is fresh}}{o(n,a,\{l|M[o'!m(\overline{v})]\},q) \to o(n,a,\{l|M[f]\},q)\ bind(C,m,\overline{v},o',f)\ f}\ (\textsc{Red-Call1})$$

$$\frac{C = classOf(o) \quad f \text{ is fresh}}{o(n,a,\{l|M[\textbf{this}!m(\overline{v})]\},q) \to o(n,a,\{l|M[f]\},q)\ bind(C,m,\overline{v},o,f)\ f}\ (\textsc{Red-Call2})$$

$$o(n,a,\{l|M[\textbf{await } g]\},q) \to o(n,a,\{l|M[\textbf{if } g \ \{\textbf{skip}\} \textbf{ else } \{\textbf{suspend}; \textbf{ await } g\}]\},q)\ (\textsc{Red-Await})$$

$$o(n,a,\texttt{idle},q)\ bound(o,pr) \to o(n,a,\texttt{idle},q \circ pr)\ (\textsc{Red-Bound})$$

$$o(n,a,\{l|M[f.\textbf{get}]\},q)\ f(v) \to o(n,a,\{l|M[v]\},q)\ f(v)\ (\textsc{Red-Get})$$

$$\frac{CT(C) = \langle n,\overline{x},\overline{I},a,mtds\rangle \quad lookup(m,\overline{v},f,mtds) = pr}{CT \rhd bind(C,m,\overline{v},o,f) \to CT \rhd bound(o,pr)}\ (\textsc{Red-Bind})$$

$$\frac{l^V(\textsf{destiny}) = f}{o(n,a,\{l|\textbf{return } v\},q)\ f \to o(n,a,\texttt{idle},q)\ f(v)}\ (\textsc{Red-Return})$$

$$\frac{cf \to cf'}{CT \rhd cf \to CT \rhd cf'}\ (\textsc{Red-Context1})$$

$$\frac{n = n' \quad CT(C) = \langle n',\overline{x},\overline{I},a',mtds\rangle}{CT \rhd o(n,a,\texttt{idle},pr \circ q) \to CT \rhd o(n,a,pr,q)}\ (\textsc{Red-Reschedule})$$

$$o(n,a,\{l|M[\textbf{suspend}]\},q) \to o(n,a,\texttt{idle},q \circ \{l|M[\textbf{skip}]\}\ (\textsc{Red-Suspend})$$

Fig. 7 The context reduction semantics.

Filling the hole of a context M with a redex r is denoted $M[r]$. Before evaluating the expression e in the method body s; **return** e, the body will be reduced to **skip**; **return** e. For simplicity, we elide the **skip** and write just **return** e.

$$\frac{n' \geq n \quad CT(C') = \langle n', \overline{x}, \overline{I}', a', mtds\rangle}{CT \rhd new(C, \overline{x}, \overline{I}, a, mtds, ((C', n) \cup dep)) \rightarrow CT \rhd new(C, \overline{x}, \overline{I}, a, mtds, dep)} \text{(Dep1)}$$

$$\frac{n' \geq n \quad CT(C') = \langle n', \overline{x}, \overline{I}', a', mtds\rangle}{CT \rhd ext(C, \overline{I}, a, mtds, ((C', n) \cup dep)) \rightarrow CT \rhd ext(C, \overline{I}, a, mtds, dep)} \text{(Dep2)}$$

$$\frac{C = classOf(o) \quad CT(C) = \langle n', \overline{x}, \overline{I}, a', mtds\rangle \quad n' > n \quad a'' = transf(a')}{CT \rhd o(n, a, \texttt{idle}, q) \rightarrow CT \rhd o(n', a'', \texttt{idle}, q)} \text{(Obj-State)}$$

$$\frac{CT(C) = \langle n, \overline{x}, \overline{I}', a', mtds'\rangle}{CT \rhd ext(C, \overline{I}, a, mtds, \emptyset) \rightarrow CT[C \mapsto \langle n+1, \overline{x}, (\overline{I}', \overline{I}), a' \circ a, mtds' \oplus mtds\rangle]} \text{(Extend-Class)}$$

$$CT \rhd new(C, \overline{x}, \overline{I}, a, mtds, \emptyset) \rightarrow CT[C \mapsto \langle 1, \overline{x}, \overline{I}, a, mtds\rangle] \text{(New-Class)}$$

$$config \xrightarrow{upd} config\ upd \text{(Update)}$$

$$\frac{rcf \rightarrow rcf'}{rcf\ cf \rightarrow rcf'\ cf} \text{(Red-Context2)}$$

Fig. 8 The context reduction semantics for dynamic software updates.

Statements. In Red-Assign1 and Red-Assign2, values are assigned to local and program variables. In Red-Cond1, the boolean condition evaluates to true so the evaluation of s is chosen. Otherwise, evaluate s' in Red-Cond2. In Red-Query1, the given interface is found among the interfaces in the class definition of an object, reducing the query to true. Otherwise, reduce to false in Red-Query2.

Expressions and Guards. In Red-Call1 and Red-Call2, external and internal asynchronous calls add a future to the configuration, and return its id to the caller. In Red-Get, the future's reply value is read. In Red-New, a new instance of class C is placed in the configuration. The new object's fields are given by the class table *CT(C)* and the active process is initiated with the `idle` process. We use $classOf(o) = C$ to retrieve the class name C from the object id o. In Red-Poll1 and In Red-Poll2, a future is polled to see if a call has been executed.

Suspension and Rescheduling. Guards determine whether a process should be suspended. In Red-Await, a process proceeds if its guard is true and suspends otherwise. When a process is suspended, its guard is reused to reschedule the process. When a process is suspended in Red-Suspend or terminates, it is replaced by the `idle` process, which allows a process from the process queue to be scheduled for execution in Red-Reschedule.

Method Calls and Returns. A method call results in an activation on the callee's process queue. There is a delay between the asynchronous call and its activation, represented by a future without any reply value and a bind request to the callee's class. In Red-Bind, the method is retrieved from the class, resulting in a *bound* message to the callee, which is loaded into the process queue in Red-Bound. When the process terminates, the result is stored by Red-Return in the future identified by the destiny variable. This future now carries a reply value and the active process becomes `idle`. Finally, Red-Context1 reduces subconfigurations.

Semantics of Dynamic Software Updates. To support updates, we consider update messages by extending the runtime syntax of Fig. 6 as follows:

$$\begin{aligned} msg &::= \ldots \mid upd \\ upd &::= new(C, \overline{x}, \overline{I}, a, mtds, dep) \mid ext(C, \overline{I}, a, mtds, dep) \\ dep &::= \overline{(C, n)} \end{aligned}$$

An update message for a new class or class extension has a class name C, class parameters $\overline{x}$ for new class, a list $\overline{I}$ of interfaces, a list a of new fields, a set *mtds* of new (or redefined) methods, and a set *dep* of constraints to classes in the runtime system. If a message injected into the runtime configuration is well-typed in an environment Γ^i, then *dep* is $\Gamma^i_d(\langle C, curr(C, \Gamma^i_d)\rangle)$. Thus, the static dependencies of the current update are introduced into the runtime configuration. The semantics for dynamic class operations consist of the rules in Fig. 8.

Dynamic software updates are initiated by injecting a message *upd* into the configuration by rule UPDATE. For the *extension* of a class C, this message is $ext(C, \overline{I}^*, a, mtds, dep)$ which cannot be applied unless the constraints in *dep* are satisfied; these are checked by DEP. Thus, the update is delayed at runtime until other updates have been applied. When the constraints are satisfied, the fields and methods of the runtime class definition are extended and the version number increased in EXTEND-CLASS. (For the operator $\oplus$, see Sect. 4.) Similarly, when the constraints are satisfied, NEW-CLASS creates a new runtime class with all implemented interfaces and its version number set to 1 (only classes in the original program have the version numbers initially set to 0).

Figure 9 illustrates the update mechanism for objects after an update has been applied to the objects' class. Fields must be updated *before* new code is allowed to execute. New instances of the class (i.e., objects created after time t_{update}) automatically get the new fields, but the update of existing objects must be controlled; if new or redefined methods were executed that rely on fields that are not yet available in the object errors may then occur. For recursive or nonterminating methods, objects cannot generally be expected to reach a state without any pending processes, even if one postpones the activation of new method calls. Waiting for the completion of all processes before applying an update is, therefore, too restrictive. However, objects may reach *quiescent* states when the processor has been released and before any pending process has been activated, which is when the active process is `idle`. Objects which do not deadlock will eventually reach such state and we have that at least one quiescent state is reached in each cycle if nonterminating activity is defined by recursion. At this point, objects can be updated. This update mechanism is asynchronous meaning that some objects may be updated before others in a non-deterministic order. The updated objects may execute a new implementation of some methods, while other objects are still running the old method implementation.

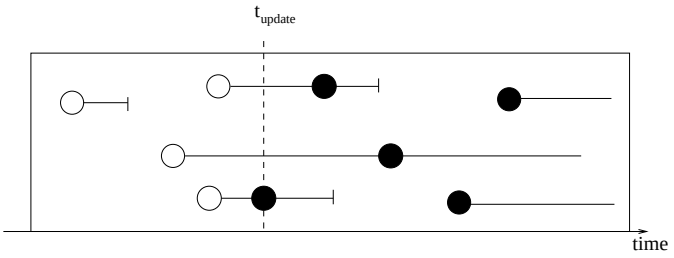

Fig. 9 Asynchronous object update.

A class update propagates to its objects in two steps. For Extend-Class, the version number of the class increases then the object gets an update the next time it interacts with its class. Before the new process is activated, the active process must become `idle`, in which case Obj-State applies (abstracting from a more asynchronous locking scheme). The *transf* function returns the new state, retaining the values of old fields. A class may be updated several times before the object reaches a quiescent state, meaning that the object may miss some updates. In this case, a single update suffices to ensure that the object is a complete instance of the present version of its class.

6 Conclusion

This paper presents a system of software updates for the asynchronous evolution of distributed, loosely-coupled active objects, motivated by IoT systems. The system emphasizes programmability, yet ensures type safety at runtime for the gradually evolving system. We have combined dynamic software updates with an interface query mechanism to facilitate context adaptation. A characteristic feature of our system is the use of type and effect analysis to identify dependencies between different updates, which are checked at runtime to ensure that the runtime view of the system is sufficiently close to the static view for updates to apply.

References

1. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems.* The MIT Press, Cambridge, Mass., 1986.
2. S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In D. Thomas, editor, *Proc. 20th European Conf. on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *LNCS*, pages 452–476. Springer, 2006.
3. T. Amtoft, F. Nielson, and H. R. Nielson. *Type and Effect Systems: Behaviours for Concurrency.* Imperial College Press, 1999.

4. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
5. E. Baccelli, J. Doerr, S. Kikuchi, F. A. Padilla, K. Schleiser, and I. Thomas. Scripting Over-The-Air: Towards Containers on Low-end Devices in the Internet of Things. In *IEEE PerCom 2018*, Mar. 2018.
6. G. Bierman, M. Hicks, P. Sewell, and G. Stoyle. Formalizing dynamic software updating. In *Proc. 2nd Intl. Workshop on Unanticipated Software Evolution (USE)*, April 2003.
7. G. Bierman, M. Parkinson, and J. Noble. UpgradeJ: Incremental typechecking for class upgrades. In *Proc. 22nd European Conf. on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 235–259. Springer, 2008.
8. F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, Oct. 2017.
9. C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In R. Crocker and G. L. S. Jr., editors, *Proc. Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, pages 403–417. ACM Press, 2003.
10. H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07. IEEE Computer Society, 2007.
11. D. Duggan. Type-Based hot swapping of running modules. In C. Norris and J. J. B. Fenwick, editors, *Proc. 6th Intl. Conf. on Functional Programming (ICFP'01)*, volume 36, 10 of *ACM SIGPLAN notices*, pages 62–73, New York, Sept. 3–5 2001. ACM Press.
12. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comp. Sci.*, 103(2):235–271, 1992.
13. D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Software Eng.*, 22(2):120–131, 1996.
14. C. M. Hayden, K. Saur, E. K. Smith, M. Hicks, and J. S. Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. *ACM Trans. Program. Lang. Syst.*, 36(4), 2014.
15. C. E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.
16. R. Hiesgen, D. Charousset, and T. C. Schmidt. Embedded actors - towards distributed programming in the IoT. In *IEEE Fourth International Conference on Consumer Electronics Berlin (ICCE-Berlin 2014)*, pages 371–375. IEEE, 2014.
17. G. Hjálmtýsson and R. S. Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proc. USENIX Tech. Conf. (USENIX '98)*, May 1998.
18. E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.
19. E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In A. Cavalcanti and D. Dams, editors, *Proc. 16th International Symposium on Formal Methods (FM'09)*, volume 5850 of *LNCS*, pages 596–611. Springer, Nov. 2009.
20. E. B. Johnsen, O. Owe, D. Clarke, and J. Bjørk. A formal model of service-oriented dynamic object groups. *Sci. Comput. Program.*, 115-116:3–22, 2016.
21. E. Latronico, E. A. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber. A vision of swarmlets. *IEEE Internet Computing*, 19(2):20–28, 2015.
22. E. A. Lee, B. Hartmann, J. Kubiatowicz, T. S. Rosing, J. Wawrzynek, D. Wessel, J. M. Rabaey, K. Pister, A. L. Sangiovanni-Vincentelli, S. A. Seshia, D. Blaauw, P. Dutta,

K. Fu, C. Guestrin, B. Taskar, R. Jafari, D. L. Jones, V. Kumar, R. Mangharam, G. J. Pappas, R. M. Murray, and A. Rowe. The swarm at the edge of the cloud. *IEEE Design & Test*, 31(3):8–20, 2014.

23. M. Lohstroh and E. A. Lee. An interface theory for the internet of things. In R. Calinescu and B. Rumpe, editors, *Proc. 13th International Conference on Software Engineering and Formal Methods (SEFM 2015)*, volume 9276 of *LNCS*, pages 20–34. Springer, 2015.
24. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In E. Bertino, editor, *Proc. 14th European Conf. on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *LNCS*, pages 337–361. Springer, June 2000.
25. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.
26. D. Miorandi, S. Sicari, F. D. Pellegrini, and I. Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.
27. A. Orso, A. Rao, and M. J. Harrold. A technique for dynamic updating of Java software. In *Proc. Intl. Conf. on Software Maintenance (ICSM'02)*, pages 649–658. IEEE Computer Society Press, Oct. 2002.
28. V. Panzica La Manna. Local dynamic update for component-based distributed systems. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE '12. ACM, 2012.
29. V. Panzica La Manna, J. Greenyer, C. Ghezzi, and C. Brenner. Formalizing correctness criteria of dynamic updates derived from specification changes. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13. IEEE Press, 2013.
30. L. Pina, L. Veiga, and M. Hicks. Rubah: Dsu for java on a stock jvm. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14. ACM, 2014.
31. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In T. D'Hondt, editor, *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010.
32. A. Sivieri, L. Mottola, and G. Cugola. Building internet of things software with ELIoT. *Computer Communications*, 89–90:141–153, 2016.
33. E. Stenberg. Key considerations for software updates for embedded linux and iot. *Linux J.*, 2017(276), Apr. 2017.
34. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. *ACM TOPLAS*, 29(4):22, 2007.
35. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
36. I. C. Yu, E. B. Johnsen, and O. Owe. Type-safe runtime class upgrades in Creol. In R. Gorrieri and H. Wehrheim, editors, *Proc. 8th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *LNCS*, pages 202–217. Springer, June 2006.