Combining Monitoring with Run-time Assertion Checking

Gouw, C.P.T. de

Concurrent Object Groups 6

In [80] Java is extended with a concurrency model based on the notion of concurrently running object groups, so-called coboxes, which provide a powerful generalization of the concept of active objects. Coboxes can be dynamically created and objects within a cobox have only direct access to the fields of the other objects belonging to the same cobox. Since one of the main requirements of the design of coboxes is a smooth integration with object-oriented languages like Java, coboxes themselves do not have an identity, e.g., all communication between coboxes refer to the objects within coboxes. Communication between coboxes is based on asynchronous method calls with standard objects as targets. An asynchronous method call spawns a local thread within the cobox to which the targeted object belongs. Such a thread consists of the usual stack of internal method calls. Coboxes support multiple local threads which are executed in an interleaved manner. The local threads of a cobox are scheduled cooperatively, along the lines of the Creol modeling language described in [55]. This means, that at most one thread can be active in a cobox at a time, and that the active thread has to give up its control explicitly to allow other threads of the same cobox to become active.

ABS (Abstract Behavioral Specification language) is a novel

language based on coboxes for modeling and analysis of complex distributed systems. It is a fully executable language with code generators for Java, Maude and Scala. In [54] a formal semantics of ABS was introduced based on asynchronous messages between coboxes. However, as of yet, no formal method for specifying and run-time verifying traces of such asynchronous messages has been developed. In this chapter, we develop tool support for the efficient run-time verification of asynchronous message passing between coboxes, independent from any backend. This latter requirement is important because in general the analysis of a particular backend is complicated by low-level implementation details. Further, it allows to generalize the analysis to all (including future) backends.

We show how to use attribute grammars extended with assertions to specify and verify (at run-time) properties of the messages sent between coboxes. To this end, we first improve the efficiency of the run-time verification tool SAGA [34], which smoothly integrates both data- and protocol-oriented properties of message sequences. Both time and space complexity of SAGA is linear in the size of the message sequence. Further we extend it to support design-by-contract for coboxes. We illustrate the effectiveness of our method by an industrial case study from the eCommerce software company Fredhopper.

6.1 Language

We formally describe coboxes by means of a modeling language which is based on the *Abstract Behavioral Specification* language [54]. We refer to our own modeling language by ACOG (pure Actor-based Concurrent Object Groups). ACOG is designed with a layered architecture, at the base are functional abstractions around a standard notion of parametric algebraic data types (ADTs). Next we have an OO-imperative layer similar to (but much simpler than) JAVA. ACOG generalizes the concurrency model of Creol [55] from single concurrent objects to concurrent object groups (coboxes). As in [80] coboxes encapsulate synchronous, multi-threaded, shared state computation on a single processor. In contrast to thread-based concurrency, task scheduling is *cooperative*, i.e., switching between tasks of the same object happens only at specific scheduling points during the execution, which are explicit in the source code and can be syntactically identified. This allows writing concurrent programs in a much less error-prone way than in a thread-based model and makes ACOG models suitable for static analysis. In fact, the standard Java concurrency model, based on threads and locks, is too low-level, error-prone and insufficiently modular for many applications areas [80]. In our dialect, unlike in [80], for simplicity we restrict to coboxes that communicate only via pure asynchronous messages, and as such form an actor-based model as initially introduced by [2] and further developed in [82].

Fig. 6.1 shows some data types and parts of interfaces used in the case study. The interface `ClientJob` models a ClientJob, the interface `Worker` models a Worker, and the interface `Coordinator` models a Coordinator. The algebraic data types (ADT) `Content` models the file system of environments in ACOG. ADTs allow specifying immutable values in functional expressions and to abstract away from implementation details such as hardware environment, file content, or operating system specifics. Specifically, `Content` is either a `File`, where an integer (e.g., its size) is taken to represent the content of a single file, or it is a directory `Dir` with a mapping of names to `Content`, thereby, modelling a file system structure with hierarchical name space. Note that an ADT may have

```
data Content = File(Int content)
             | Dir(Map<String,Content>);

interface Worker {
  Unit acceptCoordinator(Coordinator coord);
  Unit sendCurrentId(Int id);
  Unit replyRegisterItems(Bool register);
  Unit acceptItems(Set<Item> items);
  Unit acceptEntries(Set<Map<String,Content>> contents);
}

interface Coordinator {
  Unit startReplication(Worker w);
}
interface ClientJob {
  Unit registerItems(Worker w, Int id);
}
```

Figure 6.1: Data types and Interfaces

type parameters. For example, `Map` is a built-in ADT where its key and value type parameters are instantiated to `String` and `Content`.

In this subsection we describe the core constructs of our dialect of the ABS in some detail. Specifically, we describe

- algebraic data types and functions;
- interfaces
- synchronous method calls and objects creation;
- asynchronous method calls and cobox creation;
- cooperative scheduling using `await` statements.

To illustrate synchronous and asycnhronous communication we look at the implementation of how a ClientJob connects to a Worker and receives the next set of replication schedules.

Data types and Functions ACOG supports *algebraic data types* (ADT) to model data in a software system. ADTs abstract away from implementation details such as hardware environment, file content, or operating system specifics. For example in the Replication System, the following ADT `Content` models the file system of environments.

```
data Content = File(Int content)
             | Dir(Map<String,Content>);
```

ACOG supports first-order functional programming with ADT. Functional code is guaranteed to be free of side effects. One consequence of this is that functional code may not use object-oriented features. For example, the following function `isFile` checks if the given `Content` value records a file.

```
def Bool isFile(Content c) =
    case {
       File(_) => True;
             _ => False;
    };
```

Interfaces ACOG has a nominal type system with interface-based subtyping. Interfaces define types for objects. They have a name, and define a set of method signatures, that is, the names and types of callable methods. The following shows interface `Worker` that models a Worker.

```
interface Worker {
  Unit execute();
  Unit command(Command c);
  Unit acceptCoordinator(Coordinator coord);
  Unit sendCurrentId(Int id);
  Unit replyRegisterItems(Bool register);
  Unit acceptItems(Set<Item> items);
  Unit acceptEntries(Set<Map<String,Content>> contents);
}
```

Classes ACOG also supports class-based, object-oriented programming with standard imperative constructs. Classes de-

fine the implementation of objects. In contrast to Java, for example, classes do *not* define a type. Classes can implement arbitrarily many interfaces. These interfaces define the type of instances of that class. A class has to implement all methods of all its implementing interfaces. Classes are instantiated by constructors. The following class `WorkerImpl` implements `Worker`:

```
class WorkerImpl(ClientJob job,
                 SyncServer server,
                 Coordinator coord)
     implements Worker {
  Maybe<Command> cmd = Just(ListSchedule);
  WorkerImpl(ClientJob job,
             SyncServer server,
             Coordinator coord) {
       ...
  }
  Unit execute() {
       ...
  }
  Unit command(Command c) {
       ...
  }
  Unit acceptCoordinator(Coordinator coord) {
       ...
  }
  Unit sendCurrentId(Int id) {
       ...
  }
  Unit replyRegisterItems(Bool register) {
       ...
  }
  Unit acceptItems(Set<Item> items) {
       ...
  }
  Unit acceptEntries(Set<Map<String,Content>> contents) {
       ...
  }
}
```

It defines the fields `job`, `server`, `cmd` and `coord`. Those fields are typically initialized by a constructor method, or for simple initializations such as `cmd`, in the class definition itself.

Thread-based computation Basic statements describing the flow of control of a single thread include the usual (synchronous) method invocations, object creation, and field and variable reads and assignments. These statements can be composed by the standard control structures (sequential composition, conditional and iteration constructs). The following shows the part of class `ClientJobImpl` that a ClientJob connecting to a Worker and acquiring the next schedules:

```
class ClientJobImpl(SyncServer server)
     implements ClientJob {
  Unit sendSchedules(Set<Schedule> ss) {
       ...
  }
  Unit executeJob() {
       ...
  }
  Unit acceptConnection(Worker w) {
       if (w != null) {
             ...
             this.scheduleJob();
       }
  }
  Unit scheduleJobs() {
    Scheduler sr = new SchedulerImpl(...);
    sr.schedule();
  }
}
```

The method `acceptConnection` invokes synchronously the (private) method `scheduleJob`, which in turn creates an object of `SchedulerImpl` (by invoking the appropriate constructor method) and invokes its method `schedule`.

Coboxes The concurrency model of ACOG is based on the concept of *Coboxes*. A typical ACOG system consists of multiple, concurrently running coboxes. Coboxes can be regarded

as autonomous run-time components that are executed concurrently, share no state and communicate via method calls. A new object cobox is created by using the `new cog` expression. It takes as argument a class name and optional parameters and returns a reference to the initial object of the new cobox. Communication between coboxes may solely be done via *asynchronous method calls*. The difference to the synchronous case is that an asynchronous call immediately returns to the caller without waiting for the message to be received and handled by the callee. Asynchronous method calls are indicated by an exclamation mark (!) instead of a dot.

The following fragment of `ClientJobImpl` illustrates cobox creation and asynchronous communications.

```
class ClientJobImpl(SyncServer server,
                    SyncClient client,
                    Schedule s)
     implements ClientJob {

  Set<Schedule> schedules = EmptySet;

  Unit executeJob() {
       server!getConnection(this);
  }
  Unit acceptConnection(Worker w) {
       ...
  }
  Unit sendSchedules(Set<Schedule> ss) {
       ...
  }
  Unit scheduleJobs() {
       ...
  }
}

class SyncServerImpl(Coordinator coord)
     implements SyncServer {

  Unit getConnection(ClientJob job) {
       Bool shutdown = this.isShutdownRequested();
       if (shutdown) {
             job!acceptConnection(null);
```

```
      } else {
            Worker w = new cog WorkerImpl(job,
                                          this,
                                          coord);
            job!acceptConnection(w);
      }
  }
}
```

The class `SyncServerImpl` implements the SyncServer and `ClientJobImpl` implements a ClientJob. `ClientJobImpl` has a field `server` that holds the reference to the SyncServer that is assigned to a different cobox. The method `executeJob` invokes SyncServer's method `getConnection` asynchronously to connect with a Worker. In the implementation of `SyncServer`, a new object cobox is created with the `WorkerImpl` object being the initial object in that cobox.

Cooperative scheduling Each asynchronous method call results in a *task* in the cobox of the target object. Tasks are scheduled *cooperatively* within the scope of a object cobox. Cooperative scheduling means that switching between tasks of the same object cobox happens only at specific *scheduling points* during program execution and that at no point two tasks in the same cobox are active at the same time. Using the `await` statement, one can create a *conditional* scheduling point, where the running task is suspended until a Boolean condition over the object state becomes true. The following shows the implementation of `ClientJobImpl` after connecting with a `Worker`.

```
class ClientJobImpl(SyncServer server,
                    SyncClient client,
                    Schedule s)
     implements ClientJob {

  Set<Schedule> schedules = EmptySet;

  Unit sendSchedules(Set<Schedule> ss) {
       schedules = ss;
  }
  Unit acceptConnection(Worker w) {
```

```
      if (w != null) {
            w!command(Schedule(s));
            await schedules != EmptySet;
            this.scheduleJobs();
      }
  }
  ...
}

class WorkerImpl(ClientJob job,
                 SyncServer server)
     implements Worker {
  Unit command(Command c) {
      ...
      job!sendSchedules(schedules);
  }
}
```

The method `acceptConnection` invokes method `command` on the worker and suspends using the statement `await schedules != EmptySet` to wait for the next set of schedules to arrive. The next set of schedules is set by invoking the method `sendSchedules` on the ClientJob.

Fig. 6.2 shows a part of a class implementation of `Worker` that provides an implementation of method `acceptCoordinator`. The method takes a reference of the `Coordinator`. It first sets the instance variable `coord` to the input reference, it invokes the statement `await cmd != Nothing`, which suspends the current task until the side-effect free expression `cmd != Nothing` is satisfied. Instance variable `cmd` is a `Maybe` value which is either a `Command` value representing the next command to the worker from the ClientJob, or the value `Nothing` if no command has yet been given. After this, the method makes an asynchronous method call either to the `server`'s `requestListSchedules`, requesting to get all configured replication schedules, or `requestSchedule`, requesting only the schedule with the name specified by the given command. Both `fromJust` and `ssname` are functions on data types.

```
class WorkerImpl(ClientJob job,
                 SyncServer server)
     implements Worker {

  Maybe<Command> cmd = Nothing;
  Coordinator coord = null;

  Unit acceptCoordinator(Coordinator coord) {
       this.coord = coord;
       await cmd != Nothing;
       if (cmd == Just(ListSchedule)) {
             server!requestListSchedules(this);
       } else {
             server!requestSchedule(this,
                              ssname(fromJust(cmd)));
       }
  }
}
```

Figure 6.2: Method `acceptCoordinator`

6.2 Semantics

In this section we describe the formal semantics of systems of coboxes compositionally in terms of the behavior of the coboxes individually. The behavior of a cobox itself is described compositionally in terms of its threads. In this section we abstract from the functional part of the modeling language. We further abstract from variable declarations and typing information, and simply assume given a set of variables $x, y, \ldots$. We distinguish between simple and instance variables. The set of simple variables is assumed to include the special variable "this". Simple variables are used as formal parameters of method definitions.

Throughout this section we assume a given program which specifies a set of classes and a (single) inheritance relation. We start with the following basic semantic notions. For each class C we assume given a set of O_C, with typical element o, of (abstract) objects which belong to class C at run-time. A *heap* h is formally given as a set of (uniquely) labelled object

states $o : s$, where s assigns values to the instance variables of the object o. An object o exists in a heap h if and only it has a state in h, that is, $o : s \in h$, for some object state s. A heap thus represents the values of the instance variables of a group of objects. A heap is "open" in the sense that $s(x)$, for $o : s \in h$, may refer to an object that does not exist in h, i.e., that belongs to a different group. We denote $s(x)$, for $o : s \in h$, by $h(o.x)$. By s_{init} we denote the object state which results from the initialization of the instance variables of a newly created object. Further, by $h[o.x = v]$ we denote the heap update resulting from the assignment of the value v to the instance variable x of the object o. Next we introduce a *thread configuration* as a pair $\langle t, h\rangle$ consisting of a thread t. and heap h. A thread itself is a stack of closures of the form (S, τ), where S is a statement and τ is a local environment which assigns values to simple variables. By $\tau[x = v]$ we denote the update of the local environment τ resulting from the assignment of the value v to the variable x. We denote by $V(e)(\tau, h)$ the value of a side-effect free expression e in the local environment τ and global heap h. In particular we have that $V(x)(s, h) = s(x)$, for a simple variable x, and $V(x)(\tau, h) = h(\tau(\text{this}).x)$, for an instance variable x. It is important to observe that since heaps are "open" (as discussed above) $V(e)(\tau, h)$ can be undefined in case e refers to instance variables of objects that do not belong to the group represented by h.

Thread Semantics A transition

$$\langle t, h\rangle \longrightarrow \langle t', h'\rangle$$

between thread configurations $\langle t, h\rangle$ and $\langle t', h'\rangle$ indicates

- the execution of an assignment $x = e$ or
- the evaluation of a boolean condition b of an if-then-else or while statement,
- or the execution of a synchronous call.

A *labelled* transition

$$\langle t, h\rangle \xrightarrow{l} \langle t', h'\rangle$$

indicates for

$l =$ **await:** the successfull execution of an await statement,

$l = o!m(\bar{v})$**:** an asynchronous call of the method m of the object o with actual parameters $\bar{v}$.

In the following structural operational semantics for the execution of single threads $(S, s) \cdot t$ denotes the result of pushing the closure (S, s) into the stack t. We omit the transitions for sequential composition, if-then-else and while statement since they are standard.

Assignment simple variables

$$\langle (x = e; S, \tau) \cdot t, h \rangle \longrightarrow \langle (S, \tau') \cdot t, h \rangle$$

where $\tau' = \tau[x = V(e)(\tau, h)]$.
The assigment to a simple variable thus only affects the (active) local environment.

Assignment instance variables

$$\langle (x = e; S, \tau) \cdot t, h \rangle \longrightarrow \langle (S, \tau) \cdot t, h'] \rangle$$

where $h' = h[\tau(\text{this}).x = V(e)(\tau, h)]$ (assuming that $V(e)(\tau, h)$ is defined).
The assignment to an instance variable only affects the heap. Note that the assignment thus fails in case $V(e)(\tau, h)$ is undefined (such failures can be prevented by a suitable typing system, see [80]).

Await

$$\langle (\text{ await } b; S, \tau) \cdot t, h \rangle \stackrel{\text{await}}{\longrightarrow} \langle (S, \tau) \cdot t, h \rangle$$

where $V(b)(\tau, h) =$ true .
If the boolean condition of an await statement evaluates to true this transition thus additionally generates a label which will be used for synchronization with other threads (see the corresponding rule in the semantics of coboxes below).

Asynchronous method call

$$\langle (x!m(\bar{e}); S, \tau) \cdot t, h\rangle \stackrel{o!m(\bar{v})}{\longrightarrow} \langle (S, \tau) \cdot t, h\rangle$$

where $o = V(x)(s,h)$, $\bar{e} = e_1, \ldots, e_n$, $\bar{v} = v_1, \ldots, v_n$, and $v_i = V(e_i)(s,h)$, for $i = 1, \ldots, n$. An asynchronous call thus simply generates a corresponding message.

Synchronous method call

$$\langle (y = x.m(\bar{e}); S, \tau) \cdot t, h\rangle \longrightarrow \langle (S', \tau') \cdot (y = r; S, \tau) \cdot t, h\rangle$$

where, assuming that $V(x)(s,h) \in O_C$, $m(\bar{x})\{S'\}$ is the corresponding method definition in class C. Further, $\tau'(\text{this}) = V(x)(\tau, h)$ and $\tau'(x_i) = V(e_i)(\tau, h)$, for $i = 1, \ldots, n$, (here $\bar{e} = e_1, \ldots, e_n$ and $\bar{x} = x_1, \ldots, x_n$). We implicitly assume here that τ' initializes the local variables of m, i.e., those simple variables which are not among the formal parameters $\bar{x}$. Upon return for each type a distinguished simple variable r (which is assumed not to appear in the given program) will store the return value (see the transition below for returning a value).

Class instantiation

$$\langle (y = \text{ new } C(\bar{e}); S, \tau) \cdot t, h\rangle \longrightarrow \langle (y = r.C(\bar{e}); S, \tau') \cdot t, h \cup \{o' : s_{init}\}\rangle$$

where $\tau' = \tau[r = o']$, $o' \in O_C$ is a fresh object identity (i.e., not in h), where C is the type of the variable y. For each type, the distinguished variable r is used to store temporarily the identity of the new object. We implicitly assume that the constructor method returns the identity of the newly created object (by the statement "return this").

Cobox instantiation

$$\langle (y = \text{ new cog } C(\bar{e}); S, \tau) \cdot t, h\rangle \longrightarrow \langle (y = r; y!C(\bar{e}); S, \tau') \cdot t, h\rangle$$

where $\tau' = \tau[r = o']$, $o' \in O_C$ is a fresh object identity, (C is the type of the variable y). As above, the distinguished variable r is used to store temporarily the identity of the new object (here it allows to circumvent a case distinction on whether y

is a simple or an instance variable). Note that the main difference with class instantiation is that the newly created object is *not* added to the heap h and the constructor method is called asynchronously.

In contrast to [54] and [80] we allow for very flexible scheduling policies (no assumptions are made about scheduling policies at all, even for constructors, besides the fact that `await` statements are respected), it is possible that the constructor method is executed at a later stage than a normal method called on the newly created object. If this is not desired, the user can synchronize explicitly using `await`.

Return

$$\langle (\text{ return } e; S, \tau) \cdot (S', \tau') \cdot t, h\rangle \longrightarrow \langle (S', \tau'[r = v]) \cdot t, h\rangle$$

where $v = V(e)(\tau, h)$. The distinguished variable r here is used to store temporarily the return value.

In the above transitions for the creation of a class instance or a new cobox we assume a thread-local mechanism for the selection of a fresh object identity which avoids name clashes between the activated threads, the technical details of which are straightforward and therefore ommitted.

Semantics of coboxes A cobox is a pair $\langle T, h\rangle$ consisting of a set T of threads and a heap h. An object o belongs to a cobox $\langle T, h\rangle$ if and only if it has a state in h, that is, $o : s \in h$, for some object state s.

Internal computation step
An unlabelled computation step of a thread is extended to a corresponding transition of the cobox by the following rule:

$$\frac{\langle t, h\rangle \longrightarrow \langle t', h'\rangle}{\langle \{t\} \cup T, h\rangle \longrightarrow \langle \{t'\} \cup T, h'\rangle}$$

External call
A computation step labelled by an asynchronous method call

is extended to a corresponding transition of the cobox by the following rule:

$$\frac{\langle t, h\rangle \overset{o!m(\bar{v})}{\longrightarrow} \langle t', h'\rangle}{\langle \{t\} \cup T, h\rangle \overset{o!m(\bar{v})}{\longrightarrow} \langle \{t'\} \cup T, h'\rangle}$$

Synchronization
The execution of an await statement by a thread within a given cobox is formally captured by the rule

$$\frac{\langle t, h\rangle \overset{\text{await}}{\longrightarrow} \langle t', h'\rangle}{\langle \{t\} \cup T, h\rangle \longrightarrow \langle \{t'\} \cup T, h'\rangle}$$

provided all threads in T executing an await statement, that is, the top of each thread in T consists of a closure of the form $(\text{ await } b; S, \tau)$ (we implicitly assume that terminated threads are removed). Note that thus the await statement enforces a barrier synchronization of all the threads of a cobox. This synchronization ensures that at most one thread in a cobox is executing.

Input-enabledness
We further have the following transition which describes the *reception* of an asynchronous method call to an object o which belongs to the cobox $\langle T, h\rangle$:

$$\langle T, h\rangle \overset{o?m(\bar{v})}{\longrightarrow} \langle T \cup \{t\}, h\rangle$$

where, assuming that $o \in O_C$, $m(\bar{x})\{S\}$ is the corresponding method definition in class C. Further, t consists of the closure $\langle \text{ await b } ; S, \tau\rangle$, where $V(b)(\tau, h) =$ true and τ assigns the actual parameters $\bar{v}$ to the formal parameters $\bar{x}$ of m (as above, the object identity o is assigned to the implicit formal parameter "this") and initializes all local variables of m.

The added await statement enforces synchronization between the other threads. Since coboxes are *input-enabled* this transition thus models *an assumption* about the environment. This assumption is validated in the context of coboxes as described next.

Semantics of systems of coboxes Finally, a system configuration is simply a set G of coboxes. For technical convenience we assume that all system configurations contain an infinite set of *latent* coboxes $\langle \emptyset, \{o : s_{\ init}\ \}\rangle$ (for $o \in O_C$ for all classes C) which have not yet been activated. The fresh object generated by the creation of a new cobox, as described above in the thread semantics (transition 6.2), at this level is assumed to correspond to a latent cobox.

Interleaving
An internal computation step of a cobox is extended to a corresponding transition of the global system as follows.

$$\frac{g \longrightarrow g'}{\{g\} \cup G \longrightarrow \{g'\} \cup G}$$

Message passing
Communication between two coboxes is formalized by

$$\frac{g_1 \stackrel{o?m(\bar{v})}{\longrightarrow} g_1' \quad g_2 \stackrel{o!m(\bar{v})}{\longrightarrow} g_2'}{\{g_1, g_2\} \cup G \longrightarrow \{g_1', g_2'\} \cup G}$$

Here it is worthwhile to observe that for an asynchronous call $o!m(\bar{v})$ to an object o belonging to the *same* cobox there does not exist a matching reception $o?m(\bar{v})$ by a *different* cobox because coboxes have no shared objects.

Trace Semantics A trace is a finite sequence of input and output messages, e.g., $o?m(\bar{v})$ and $o!m(\bar{v})$, respectively. For each coboxes g we define its trace semantics $T(g)$ by

$$\{\langle \theta, g' \rangle \mid \ g \stackrel{\theta}{\longrightarrow} g'\}$$

where $\stackrel{\theta}{\longrightarrow}$ denotes the reflexive, transitive closure of the above transition relation between coboxes, collecting the input/output messages. Note that the trace θ by which we can obtain from g a cobox g' does *not* provide information about object creation or information about which objects belong to the same cobox. In fact, information about which objects have been

created can be inferred from the trace θ. Further, in general a cobox does not "know" which objects belong to the same cobox.

The following compositionality theorem is based on a notion of *compatible* traces which roughly requires for every input message a corresponding output message, and vice versa. We define this notion formally in terms of the following rewrite rule for sets of traces

$$\{o?m(\bar{v}) \cdot \theta, o!m(\bar{v}) \cdot \theta'\} \cup \Theta \Rightarrow \{\theta, \theta'\} \cup \Theta$$

This rule identifies two traces in the given set which have two matching initial messages which are removed from these traces in the resulting set. Note that this identification is nondeterministic, i.e., for a given trace there may be several traces with a matching initial message. A set of traces Θ is compatible, denoted by $\mathit{Compat}\ (\Theta)$, if we can derive the singleton set $\{\epsilon\}$ (ϵ denotes the empty trace). Formally, $\mathit{Compat}\ (\Theta)$ if and only if $\Theta \Rightarrow^* \{\epsilon\}$, where $\Rightarrow^*$ denotes the reflexive, transitive closure of $\Rightarrow$.

Theorem 6.2.1 *Let $\rightarrow^*$ denote the reflexive, transitive closure of the above transition relation between system configurations. We have*

$$G \longrightarrow^* G'$$

if and only if $G = \{g_i \mid i \in I\}$ and $G' = \{g'_i \mid i \in I\}$, for some index set I such that for every $i \in I$ there exists $\langle \theta_i, g'_i \rangle \in T(g_i)$, with $\mathit{Compat}\ (\{\theta_i \mid i \in I\})$.

Proof: The proof is straightforward but tedious and proceeds by induction on the derivation.

The above theorem states that the overall system behavior can be described in terms of the above trace semantics of the individual coboxes. This means that for compositionality no further information is required. Next we show in the following section how to specify properties of the externally observable behavior of a cobox, as defined by its traces of input/output messages.

6.3 Behavioral Interfaces for Coboxes

In this section we use the previously introduced *attribute grammars* extended with assertions to specify and verify properties of the traces generated between coboxes. As such, extended attribute grammars provide a new formalism for contracts in general, and coboxes in particular. In contrast to classes or interfaces, coboxes are run-time entities which do not have a single fixed interface[1]. In particular, newly created objects of any type can be added dynamically at any time to a group by executing a `new`-statement without the `cog` keyword. The execution of such a statement expands the group with the new object, which can be of a type different from all the objects in the group so far and consequently provides methods not provided by the other objects in the group.

As a crude approximation of the behavior of a group, we could just take the behavior of the class `C` of the object which is created by a `new cog C` statement. Such an approach requires no modification in specification languages (and corresponding tools) traditionally used for object-oriented languages (be it assertions on states, or trace-based specifications). However this basically corresponds to the assumption that all groups contain only a single object, bypassing the very concept of *groups* of objects and essentially resulting in concurrent objects, not concurrent object *groups*. This can be partly alleviated by assuming that besides the object of type `C`, objects of other types can also be part of the group by having specifications of the form 'if an object of type `D` is part of the group, then ϕ'. However, this means that *all* groups whose first created object is of type `C` must satisfy the property (since the group name in this case does not depend on other objects in the group). We abandon this idea in favor of a more fine-grained specification of groups. We first discuss how we can still refer statically, in the program text, to these run-time entities by communication views.

[1] We consider interfaces here to be a list of all signatures of the methods supported by some object in the cobox

Communication Views for COGs

To be able to refer to coboxes in syntactical constructs (such as specifications), we introduce the following (optional) annotation of cobox instantiations:

$$S \quad ::= \quad y = \text{new cog [Name]} \;\; C(\bar{e})$$

The semantics of the language remain unchanged. Note that the same cobox name can be shared among several coboxes (i.e. is in general not unique) since different cobox creation statements can specify the same cobox name. First, the same group creation statement `y = new cog A C` can appear multiple times in the program. Second, a group creation statement can be surrounded by a loop or recursive method, causing it to be executed multiple times. In both cases, all groups created by the statement receive the same name. However in contrast to the previously sketched abandoned proposal (where different named groups are distinguished at the class level of the first object in the group), in this approach, named groups are distinguished at the finer-grained statement level.

Now that we can refer to groups syntactically, the question arises what kind of specification languages would be suitable to specify the behavior of a named group. Groups communicate with other groups exclusively using asynchronous method calls, and there are no returns. Thus, the behavior of a group as observed by its environment (other groups) is simply a sequence of asynchronous method calls sent and received by objects in the group. Taking the set of all legal sequences of asynchronous method calls (also known as traces) of the group as its specification is a natural choice. We observe asynchronous method calls directly when the method call statement executes in the group, not when the actual method body of the called method begins to execute (note that the latter can happen at a much later time in our concurrency model, or even not at all in the absence of fairness assumptions). This convention allows an orthogonal treatment of scheduling policies. However, we have to face the following problem: Coboxes do not have a fixed interface, as the methods which can be invoked on an object in a cobox (and consequently appear in traces) are not fixed statically. In particular, during execution objects of any type

can be added to a cobox, which clearly affects the possible traces of the cobox. Additionally, for practical reasons it is often convenient to focus on a particular subset of methods, leaving out methods irrelevant for specification purposes. This is especially useful for incomplete specifications. To solve both these problems, we use *communication views*. A communication view can be thought of as an interface for a named cobox. Figure 6.3 shows an example communication view associated with all coboxes named WorkerGroup. Formally a communi-

```
view WorkerView grammar Worker.g specifies WorkerGroup {
  send Coordinator.startReplication(Worker w) st,
  send ClientJob.registerItems(Worker w, Int id) pr,
  receive Worker.sendCurrendId(Int id) id,
  receive Worker.replyRegisterItems(Bool reg) ar,
  receive Worker.acceptItems(Set<Item> items) is,
  receive Worker.acceptEntries(
           Set<Map<String, Content>> contents) es
}
```

Figure 6.3: Communication View

cation view is a partial mapping from messages to abstract event names. A communication view thus simply introduces names tailored for specification purposes (see the next subsection about grammars for more details on how this event name is used). Partiality allows the user to select only those asynchronous methods relevant for specification purposes. Names (such as 'st' for the method `startReplication`) are not strictly needed, but can be used to identify calls to different methods. Any method not listed in the view will be irrelevant in the specification of WorkerGroups.

Note that in this asynchronous setting we can distinguish three different events: sending a message (at the call-site), receiving the message in the queue (at the callee-site), and scheduling the message for execution (i.e. the point in time when the corresponding method starts executing). By the asynchronous nature of the ABS, we cannot detect in the ABS itself when a message has been put into the queue. Therefore we restrict to the other two events. Since we implement the run-time checker independently from any back-end

(see also Section 6.4), we are forced to use the ABS itself for the detection of the observable events. The `send` keyword identifies calls from objects in the WorkerGroup to methods of objects in another cobox (in other words: methods required by an object in the WorkerGroup). Conversely, the keyword `receive` identifies the scheduling of calls from another cobox to an object in a WorkerGroup. It is possible that methods listed in the view actually can never be called in practice (and therefore won't appear in the local trace of a cobox). For example, if WorkerGroups are created by a statement `y = new cog WorkerGroup Worker2`, only objects of the class `Worker2` are guaranteed to be part of the group. Thus messages of the form `receive Worker.*` can only be received in those WorkerGroups in which a `Worker`-object was added. The introduction of names for messages gives rise to a small refinement of our notion of a specification of a group. A specification is not a set of sequences of asynchronous method calls anymore, instead a specification is a set of sequences of names.

Communication views allow the selection of messages essentially just on the basis of the method name. But messages also involve and contain data: they are sent between an object in one group (the caller), to an object in another group (the callee) with the actual parameter values as the contents of the message. Thus the question arises what data (caller, callee, actual parameters) we can observe and use in specifications of groups. Clearly the parameter values sent in a message influence the behavior of the group which receives the message. On the other hand, as can be seen by inspecting the formal (compositional) semantics introduced in the previous section, the identity of the caller of a `receive` message does not influence the behavior of a group. In particular, there is no way to detect whether two messages originate from the same group (or even the same object). Thus it would be unnatural if one could refer in a specification to the identity of the caller: this results in specifications that cannot be satisfied by any implementation. Consequently we disallow any reference to the identity of the caller in specifications, and take the callee and the actual parameter values as the only data that can be observed from a message. A fully abstract semantics allows to determine the

minimum amount of information that needs to be captured. The introduction of data introduces another refinement of our notion of a specification. Message names are not just strings anymore, they also contain the identity of the callee and the actual parameter values. A specification for a group is still a set of the legal sequences of names (as above), but since names now also contain the callee and parameter values, their values can be restricted by the specification. Note that specifications combine protocol-oriented properties (such as the legal orderings between messages) and data-oriented properties (such as the allowed parameter values). The next subsection discusses how specifications can be defined syntactically in a convenient way.

We are now in the position to formally define when an implementation satisfies a specification.

Definition Let P be a program and S a specification (set of traces). Then $P \models S$ iff For all sets of groups $G = \{g_i \mid \ i \in I\}$: For all $i \in I : \theta_i \in T(g_i)$ and compat$(\{\theta_i | i \in I\})$ implies $\text{Proj}_V(\theta_i) \in S$.

In this definition we assume a mechanism $\text{Proj}_V(\theta_i)$for projecting each trace of a named group on the events listed in the associated communication view V. Informally the definition says that a P satisfies S if the traces of P are a subset of those of S.

Attribute Grammars

In this subsection we describe how properties of the set of allowed traces of a cobox can be specified in a convenient, high-level and declarative manner. We illustrate our approach by partially specifying the behavior depicted by the UML sequence diagram in Figure 6.4. Informally the property we focus on is:

> The Worker first notifies the Coordinator its intention to commence a replication session, the Worker would then receive the last transaction id identifying the version of the data to be replicated, the Worker sends this id to the ClientJob to see if the

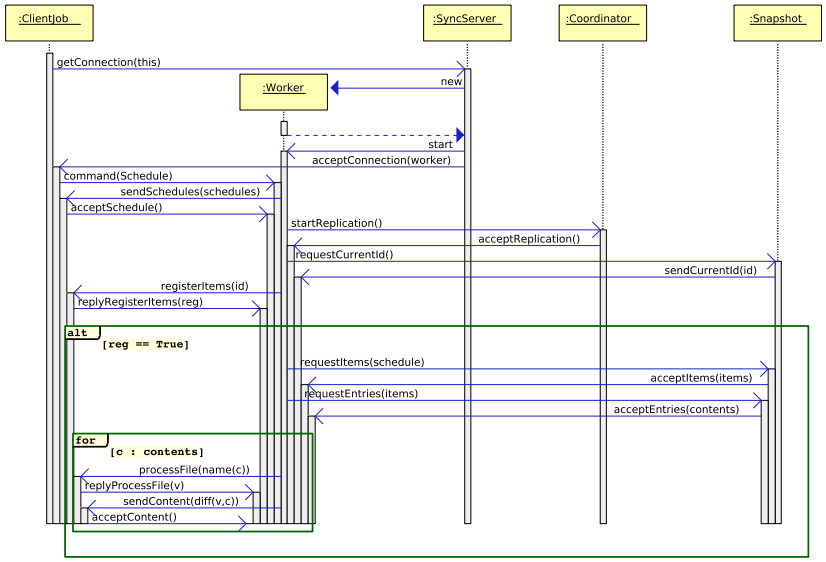

Figure 6.4: Replication interaction

client is required to update its data up to the specified version. The Worker then expects an answer. Only if the answer is positive can the Worker retrieve replication items from the snapshot, moreover, the number of files sets to be replicated to the ClientJob must correspond to the number of replication items retrieved.

The formalization of the above property uses the communication view depicted in Figure 6.3. The productions of the grammar underlying the attribute grammar in Figure 6.5 specify the legal orderings of these messages named in the view. For example, the productions

$$\begin{array}{lcl} S & ::= & \epsilon \\ & | & st\ \ T \\ T & ::= & \epsilon \\ & | & id\ \ U \end{array}$$

specify that the message 'id' is preceded by the message 'st'.

The grammars above specify only the *protocol structure* of the valid traces, but do not take the data-flow into account. To that end, we extend the grammar with attributes and assertions over these attributes. Each terminal symbol has *built-in* attributes consisting of the parameter names for referring to the object identities of the actual parameters, and `callee` for referencing the identity of the callee (see the end of the previous subsection for a motivation to include these particular attributes). Non-terminals have *user-defined* attributes to define data properties of sequences of terminals. In each production, the value of the attributes of the non-terminals appearing on the right-hand side of the production is defined.[2] For example, in the following production, the attribute 'w' for the non-terminal 'T' is defined.

$$\begin{array}{lcll} S & ::= & \epsilon & \\ & | & st \;\; T & (T\texttt{.w} = st\texttt{.w;}) \end{array}$$

Attribute definitions are surrounded by '(' and ')'. However the attributes themselves do not alter the language generated by the attribute grammar, they only *define* properties of data-flow of the trace. We extend the attribute grammar with assertions to specify properties of attributes. For example, the assertion in the second production of

$$\begin{array}{lcll} T & ::= & \epsilon & \\ & | & id \;\; U & (U\texttt{.w} = T\texttt{.w; } U\texttt{.i} = id\texttt{.id;}) \\ U & ::= & \epsilon & \\ & | & pr & \{\texttt{assert } U\texttt{.w} == pr\texttt{.w} \\ & & & \quad \texttt{\&\& } U\texttt{.i} == pr\texttt{.id;}\}\ V \end{array}$$

expresses that the 'id' passed as a parameter to the method 'registerItems' (represented in the grammar by the terminal pr`.id`) must be the same as the one previously passed into 'sendCurrentId' (terminal id`.id`). Assertions are surrounded by '{' and '}' to distinguish them visually from attribute definitions.

[2] In the literature, such attributes are called inherited attributes.

S	$::= \epsilon$	$\mid st$	T (T`.w =` st`.w;`)
T	$::= \epsilon$	$\mid id$	U (U`.w =` T`.w;` U`.i =` id`.id;`)
U	$::= \epsilon$	$\mid pr$	{`assert` U`.w ==` pr`.w` `&&` U`.i ==` pr`.id;`} V
V	$::= \epsilon$	$\mid ar$	W (W`.b =` ar`.reg;`)
W	$::= \epsilon$	$\mid is$	{`assert` W`.b;`} X (X`.s = size(`is`.items);`)
X	$::= \epsilon$	$\mid es$	{`assert` X`.s == size(`es`.contents);`}

Figure 6.5: Attribute Grammars

The full attribute grammar Figure 6.5 formalizes the informal property stated in the beginning of this subsection. The grammar specifies that for each Worker object, in its own object cobox, the Coordinator must be notified of the start of the replication by invoking its method `startReplication` (st). Only then can the Worker receive (from an unspecified cobox) the identifier of the current version of the data to be replicated (id). Next the Worker invokes the method `registerItems` on the corresponding ClientJob about this version of the data (pr). The grammar here asserts that the identifier is indeed the same as that received via the method call `sendCurrendId`. The Worker then expects to receive a method call `replyRegisterItems` indicating if the replication should proceed, the Worker then can recieve method call `acceptItems` for the data items to be replicated. The grammar here asserts that this can only happen if the previous call indicated the replication should proceed. The Worker then can receive method call `acceptEntries` for the set of Directories, each identified by a data item. Since each data item refers to a directory, the grammar here asserts the number of items is the same as the number of directories.

To further illustrate the above concepts, we consider an additional behavioral interface for the WorkerGroup cobox. To allow users to make changes to the replication schedules during the run-time of FAS, every ClientJob would request the next set of replication schedules and send them to SyncClient for scheduling. Here is an informal description of the property, where Figure 6.6 presents the communication view capturing

```
view ScheduleView grammar Schedule.g
specifies WorkerGroup {
  receive Worker.command(Command c) cm,
  send ClientJob.sendSchedules(Set<Schedule> ss) sn,
  send SyncServer.requestListSchedules(Worker w) lt,
  send SyncServer.requestSchedule(Worker w, String name)
      gt,
  send Coordinator.requestStartReplication(Worker w) st
}
```

Figure 6.6: Communication View for Scheduling

$$
\begin{array}{llll}
S & ::= \epsilon & \mid cm & T\ (T.\texttt{c} = cm.\texttt{c};) \\
T & ::= \epsilon & \mid gt & \{\texttt{assert}\ T.\texttt{c}\ \texttt{!=}\ \texttt{ListSchedule}\ \texttt{\&\&} \\
 & & & \quad gt.\texttt{n} == \texttt{name}(T.\texttt{c});\}\ U\ (U.\texttt{c} = T.\texttt{c};) \\
 & & \mid lt & \{\texttt{assert}\ T.\texttt{c} == \texttt{ListSchedule};\}\ U\ (U.\texttt{c} = T.\texttt{c};) \\
U & ::= \epsilon & \mid sn & \{\texttt{assert}\ sn.\texttt{ss}\ \texttt{!=}\ \texttt{EmptySet};\}\ V\ (V.\texttt{c} = U.\texttt{c};) \\
V & ::= \epsilon & \mid st & \{\texttt{assert}\ V.\texttt{c}\ \texttt{!=}\ \texttt{ListSchedule};\}
\end{array}
$$

Figure 6.7: Attribute Grammar for Scheduling

the relevant messages and Figure 6.7 presents the grammar that formalizes the property:

> A ClientJob may request for either all replication schedules or a single schedule. The ClientJob does this by sending a command to the Worker (cm). If the command is of the value `ListSchedule`, the Worker is to acquire all schedules from the SyncServer (lt) and return them to the ClientJob (sn). Otherwise, the Worker is to acquire only the specified schedule (gt) and return it to the ClientJob (sn). If the ClientJob asks for all schedules, it must not proceed further with the replication session and terminate (st).

In summary, a communication view provides an interface of a named cobox. The behavior of such an interface is specified

by means of an attribute grammar extended with assertions. This grammar represents the legal traces of the named cobox as words of the language generated by the grammar, which gives rise to a natural notion of the satisfaction relation between programs and specifications. Properties of the control-flow and data-flow are integrated in a single formalism: the grammar productions specify the valid orderings of the messages (the control-flow of the valid traces), whereas assertions specify the data-flow.

6.4 Implementation

In this section we discuss the architecture of the run-time checker for coboxes, identify crucial design decisions and its performance. The cobox version of SAGA is implemented as a run-time checker for ABS models. ABS is basically an extension of the modeling language considered in this paper. It is tool-supported by various analysis tools [88] and automated code generation has been implemented to various lower-level languages including Java, Maude and Scala. SAGA tests whether an actual execution of a given ABS model satisfies its specification given by attribute grammars, and stops the running program in case of a violation to prevent unsafe behavior. It is implemented as a meta-program in Rascal. The full meta-program consists of approximately 1100 lines of code.

Design The design of the cobox version of SAGA was guided by several requirements.

1. *All* back-ends (even future ones) which generate code from ABS models to lower-level target languages should be supported, without having to update SAGA when any of the back-ends is updated (for example, to generate more efficient code). Consequently we need a parser-generator which generates ABS code, and therefore cannot use existing parser generators.

2. The overhead induced by SAGA must be kept to a minimum. In particular, whenever the trace of a cobox is updated with a new message, SAGA should be able to decide *in constant time* whether the new trace still satisfies the specification (the attribute grammar). This is determined by parsing the trace (then considered as a sequence of tokens) in a parser for the attribute grammar.

3. Because of the intrinsic complexity of developing efficient and user-friendly parser generators, we require that the implementation of the parser-generator should be decoupled from the rest of the implementation of SAGA.

These requirements are far from trivial to satisfy. For example JML, a state-of-the-art specification language for Java,

has no stable version of the run-time checker which supports all back-ends (and future ones) for Java, violating the first requirement. This is due to the fact that the JML run-time checker was designed as an extension of a proprietary Java compiler. Other tools for run-time verification such as MOP and LARVA satisfy the requirement to a certain extent. Their implementation is based on AspectJ, a compiler which extends Java with aspect-oriented programming. AspectJ can transform Java programs in bytecode form. Hence all back-ends which generate bytecode compatible with AspectJ are also supported by MOP and LARVA. This includes most, though not all, versions of the standard Sun Java compiler. However aspect-oriented programming is currently not supported by the ABS. We choose an approach based on pre-processing. Specifications (consisting of a communication view and attribute grammar) are not added to the formal syntax of the programming language, they are put in separate files. This avoids creating multiple branches of the ABS language. In JML, specifications are added to the actual source, but in comments (so they are not part of the "logic" of the program). In MOP and LARVA, specifications are also separated from the programming language.

The input of SAGA consists of three ingredients: a communication view, an attribute grammar extended with assertions and an ABS model. The output is an ordinary ABS model which behaves the same as the input program, except that it throws an assertion failure when the current execution violates the specification. Since the resulting ABS model is an ordinary ABS model, all analysis tools[88] (including a debugging environment with visualization and a state-of-the-art cost analyzer) and back-ends which exist for the ABS can be used on it directly. The third requirement (a separation of concerns between the parser-generator and the rest of the implementation) has lead to a component-based design consisting of a parser-generator component and source-code weaving component. We discuss these components, and the second requirement on performance of the generated parser, in more detail below.

Parser generator component The parser-generator component processes only the attribute grammar and generates a

parser for it, with ABS as the target language. Parsers for attribute grammars in general take a stream of terminals as input, and output a parse tree according to the grammar productions (where non-terminal nodes are annotated with their attribute values). In our case, the attribute grammars also contains assertions, and the generated parser additionally checks that all assertions in the grammar are true.

In our case, whenever a new message (asynchronous call) is added to the trace, all parse trees of all prefixes have been computed previously. The question arises how efficient the new parse trees can be computed by exploiting the parse trees of the prefixes. Unfortunately, for general context-free grammars, this cannot be done in constant time using currently known algorithms (violating the second requirement on performance). For if this was possible in constant time, parsing the full trace results in a parser which works in linear time (n terminals which all take a constant amount of time), and no linear time algorithm for general context-free grammars is known. We therefore restrict to deterministic regular attribute grammars with only inherited attributes. All grammars used in the case study have this form and parsing the new trace in such grammars can be done in constant time, since they can be translated to a finite automaton with conditions (assertions) and attribute updates as actions to execute on transitions. Parsing the new message consists of taking a single step in this automaton. Moreover for such grammars, the space complexity is also very low: it is not necessary to store the entire trace, only the attribute values of the previous trace must be stored.

Source-code weaving component The weaving component processes the communication view and the given ABS model, and outputs a new ABS model in which each call to a method appearing in the view is transformed. The transformation checks whether the method call which is about to be executed is allowed by the attribute grammar, and if this is not the case, prevents unsafe behavior by throwing an assertion failure. This transformation is invasive, in the sense that it cannot be done only locally in the body of those methods actually appearing in the view, but instead it has to be

done at all call-sites (in client code). To see this, suppose that the transformation *was* done locally, say in the beginning of the method body. Due to concurrency and scheduling policies, other methods which were called at a later time could have been scheduled earlier. In such a scenario, these other methods are checked earlier than the order in which they are actually called by a client, which violates the decision (see also the previous section) to treat scheduling policies orthogonally.

The transformation is done in two steps. First, all calls to methods that occur in a communication view are isolated using pattern matching in the meta-program. We created a Rascal ABS grammar for that purpose. The ABS grammar contains around 240 non-terminals: for comparison, the Java grammar in Rascal has about 120. The main reason for the significantly larger size is that the ABS contains an internal sublanguage (for feature models and delta programming [79, 23]) for designing software product lines. The following snippet from the Rascal ABS grammar descibes the syntax for asynchronous method calls (i.e. `om(e1, ..., en)!`).

```
syntax AsyncCall
       = PureExpPrefix ! IDENTIFIER ( DataExp ","* );
```

In the second step, all asynchronous call-statements are preceded by code which checks that the current object is part of a named cobox (note that this check really has to be done at run-time due to the dynamic nature of coboxes). If this is the case, the trace is updated by taking a step in the finite automaton where additionally the assertion is checked. If there is no transition for the message from the current state, we throw an assertion error. Intuitively such an error corresponds to a protocol violation. There is one subtle point about updating the trace. If no assumptions are made about the scheduling of received messages, only updates to the trace of the calling cobox (i.e. 'send' messages in the view) can be guaranteed to be executed directly before the actual call happens. For 'receive' messages the history is updated whenever the corresponding method begins executing. Thus this assymmetry between 'send' and 'receive' events is natural when one takes into account that the actual behavior of the program only depends on the order in which the 'receive' events (or rather, the associated methods) are actually executed.