Shacl: Operational Semantics

Joeri De Koster, Tom Van Cutsem

Vrije Universiteit Brussel,
Pleinlaan 2,
B-1050 Brussels, Belgium

1. Operational semantics

In this section we describe the operational semantics for a small but significant subset of our language named SHACL-LITE. The operational semantics serves as a reference specification of the semantics of our language abstractions regarding domains and views. The operational semantics of SHACL-LITE was primarily based on an operational semantics for the AmbientTalk language [1] which in turn was based on that of the Cobox [2] model. Our operational semantics models actors, objects, event-loops and domains.

$K \subseteq$ **Configuration**	::=	$\mathcal{K}\langle A, D, R\rangle$	Configurations
$a \in A \subseteq$ **Actor**	::=	$\mathcal{A}\langle \iota_a, S, E, Q, e\rangle$	Actors
$d \in D \subseteq$ **Domain**	::=	$\mathcal{D}\langle \iota_d, l, O\rangle$	Domains
$R \subseteq$ **Request**	::=	$\mathcal{R}\langle \iota_a, \iota_d, t, e\rangle$	View Requests
$o \in O \subseteq$ **Object**	::=	$\mathcal{O}\langle \iota_o, F, M\rangle$	Objects
$m \in$ **Message**	::=	$\mathcal{M}\langle r, m, \overline{v}\rangle$	Messages
$n \in N \subseteq$ **Notification**	::=	$\mathcal{N}\langle \iota_d, t, e\rangle$	Notifications
$Q \subseteq$ **Queue**	::=	$\overline{m \mid n}$	Queues
$M \subseteq$ **Method**	::=	$\mathrm{m}(\overline{x})\{e\}$	Methods
$F \subseteq$ **Field**	::=	$f : v$	Fields
$v \in$ **Value**	::=	$r \mid t \mid \mathrm{null}$	Values
$r \in$ **Reference**	::=	$\iota_d.\iota_o$	References
$t \in$ **RequestType**	::=	$\textsc{s} \mid \textsc{e}$	Request Types
$l \in$ **AccessModifier**	::=	$\textsc{r}(n) \mid \textsc{w} \mid \textsc{f}$	Access Modifiers
$\iota_d \in S, E \subseteq$ **DomainId**			Domain Identifiers
$i \in \mathbb{N}$			Integers

$$\iota_o \in \textbf{ObjectId}, \iota_d \in \textbf{DomainId}, \iota_a \in \textbf{ActorId}, \textbf{ActorId} \subseteq \textbf{DomainId}$$

Figure 1: Semantic entities of SHACL-LITE.

1.1. Semantic entities

Figure 1 lists the different semantic entities of SHACL-LITE. Calligraphic letters like $\mathcal{A}$ and $\mathcal{M}$ are used as "constructors" to distinguish the different semantic entities syntactically. Actors, domains, and objects each have a distinct address or identity, denoted ι_a, ι_d and ι_o respectively.

Email addresses: jdekoste@vub.ac.be (Joeri De Koster), tvcutsem@vub.ac.be (Tom Van Cutsem)

In SHACL-LITE a **Configuration** consists of a set of live actors, A, a set of domains, D, and a set of pending view requests, R. A single configuration represents the whole state of a SHACL-LITE program in a single step. In SHACL-LITE each **Actor** has an identity ι_a. Similarly, each **Domain** has an identity ι_d. All actors are associated with a single domain with the same identity as the actor. Domains can also be created by an actor without being associated with that actor. This domain represents the actor's heap. This design decision makes it so that all objects belong to a certain domain. Meaning that accessing these objects can be uniformly defined. Because each actor is associated with a single domain (the one with the same Id as the actor, i. e., $\iota_a = \iota_d$), the set of actor Ids is a subset of the set of domain Ids. Each actor has two sets of domain Ids, S and E, on which it currently holds respectively shared and exclusive access. Upon the creation of an actor that actor always has exclusive access to its associated domain. This means that the set E initially contains the singleton identifier ι_a. An actor also has a queue of pending messages Q, and the expression e it is currently evaluating, i. e., reducing. While each actor is associated with a single domain, the inverse does not hold. In addition to an identity, a domain also has a single **Access Modifier** l (or lock) and a set of objects O, that represents the object heap of that domain. A pending **Request** has a reference ι_a, to the actor that placed the request, a reference to the domain ι_d on which the view was requested, the type of request and an expression e, that will be reduced in the context of a view once the domain becomes available. The **Type** of a request is either shared (S) or exclusive (E). Note that the way views are assigned to actors implements a simple multiple-reader, single-writer locking strategy. This limits the expressiveness of our implementation in favor of more concise abstractions. An **Object** has an identity ι_o, a set of fields F, and a set of methods M. An asynchronous **Message** holds a reference r, to the object that was the target of the message, the message identifier m, and a list of values $\overline{v}$, that were passed as arguments. A **Notification** or view is a special type of event that has a reference to the domain on which a view was requested, the type of view that was requested and the expression that is to be reduced once the notification-event is being processed. The **Queue** used by the event-loop of an actor is an ordered list of pending messages and notifications. A **Method** has an identifier m, a list of parameters $\overline{x}$, and a body e. A **Field** consists of an identifier f, that is bound to a value v. **Values** can either be a reference r, a request type or `null`.

1.2. SHACL-LITE *syntax*

Syntax. SHACL-LITE features both functional as well as object-oriented elements. It has anonymous functions ($\lambda x.e$) and function invocation ($e(\overline{e})$). Local variables can be introduced with a let statement. Objects can be created with the `object` literal syntax. Objects may be lexically nested and are initialized with a number of fields and methods. Those fields can be updated with new values and the object's methods can be called both synchronously ($e.m(\overline{e})$) as well as asynchronously ($e \leftarrow m(\overline{e})$). In the context of a method, the pseudovariable `this` refers to the enclosing object. `this` cannot be used as a parameter name in methods or redefined using `let`.

New domains can be created using the `domain` literal. This creates a new object with the given fields and methods in a fresh domain. New actors can be spawned using the `actor` literal expression. Similarly to domains, this creates a new object with the given fields and methods in a fresh domain. This domain is then linked with a fresh actor by sharing the same identifier. The newly created actor executes in parallel with the other actors in the system. Expressions contained in domain and actor literals may not refer to lexically enclosing variables, apart from the this-pseudovariable.That is, they must have $FV(e) \subseteq \{$ `this` $\}$ for all field initializer and method body expressions e. Actors and domains are isolated from their surrounding lexical scope, making them self-contained.

SHACL's `whenShared` and `whenExclusive` primitives are represented by the $\text{acquire}_e(e)\{e\}$ primitive in SHACL-LITE. The `aquire` primitive is used to acquire views on a domain. It is parametrized with three expressions of which the first two have to reduce to a request type and a domain identifier respectively.

Syntactic sugar. Anonymous functions are translated to objects with one method named `apply`. Note that the pseudovariable `this` is replaced by a newly introduced variable x_{this} such that `this` still references the

Shacl-Lite Syntax

$$
\begin{array}{rcl}
e \in E \subseteq \textbf{Expression} & ::= & \text{this} \mid x \mid t \mid \text{null} \mid e\,;\, e \mid \lambda x.e \mid e(\overline{e}) \mid \text{let } x = e \text{ in } e \mid e.f \mid e.f := e \\
& \mid & e.m(\overline{e}) \mid \text{actor}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \mid \text{object}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \\
& \mid & \text{domain}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \mid e \leftarrow m(\overline{e}) \mid \text{acquire}_t(e)\{e\} \\
x, x_f, x_r \in \textbf{VarName}, f \in \textbf{FieldName}, m \in \textbf{MethodName} &&
\end{array}
$$

Syntactic Sugar

$$
\begin{array}{rcll}
e\,;\, e' & \stackrel{\text{def}}{=} & \text{let } x = e \text{ in } e' & x \notin \text{FV}(e') \\
\lambda x.e & \stackrel{\text{def}}{=} & \text{let } x_{this} = \text{this in} & x_{this} \notin \text{FV}(e) \\
 & & \quad \text{object } \{ & \\
 & & \quad\quad \text{apply}(x)\{[x_{this}/\text{this}]e\} & \\
 & & \quad \} & \\
e(\overline{e}) & \stackrel{\text{def}}{=} & e.\text{apply}(\overline{e}) & \\
\text{whenShared}(e)\{e'\} & \stackrel{\text{def}}{=} & \text{acquire}_{\text{S}}(e)\{e'\} & \\
\text{whenExclusive}(e)\{e'\} & \stackrel{\text{def}}{=} & \text{acquire}_{\text{E}}(e)\{e'\} &
\end{array}
$$

Evaluation Contexts and Runtime Syntax

$$
\begin{array}{rcl}
e_\Box & ::= & \Box \mid \text{let } x \;=\; e_\Box \text{ in } e \mid e_\Box.f \mid e_\Box.f := e \mid v.f := e_\Box \mid e_\Box.m(\overline{e}) \mid v.m(\overline{v}, e_\Box, \overline{e}) \mid \\
 & & e_\Box \leftarrow m(\overline{e}) \mid v \leftarrow m(\overline{v}, e_\Box, \overline{e}) \mid \text{acquire}_{e_\Box}(e)\{e\} \mid \text{acquire}_v(e_\Box)\{e\} \\
 & & \\
e & ::= & \ldots \mid r \mid \text{release}_{\text{t}}(\text{v}) \mid \text{object}_{\iota_d}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \mid r.f : e
\end{array}
$$

surrounding object in the body-expression of that anonymous function. Applying an anonymous function is the same as invoking the method `apply` on the corresponding object.

Evaluation contexts and runtime syntax. We use evaluation contexts [3] to indicate what subexpressions of an expression should be fully reduced before the compound expression itself can be further reduced. $e_\Box$ denotes an expression with a "hole". Each appearance of $e_\Box$ indicates a subexpression with a possible hole. The intent is for the hole to identify the next subexpression to reduce in a compound expression.

Our reduction rules operate on "runtime expressions", which are simply all expressions including references r, the special primitive `release` generated by reducing an acquire statement, object literals that are annotated with the domain identifier of their lexically enclosed domain and field initialization. This annotation is required such that upon object creation each object gets associated with the appropriate domain.

1.3. Reduction rules

Notation. Actor heaps O are sets of objects. To lookup and extract values from a set O, we use the notation $O = O' \uplus \{o\}$. This splits the set O into a singleton set containing the desired object o and the disjoint set $O' = O \setminus \{o\}$. The notation $Q = Q' \cdot \text{m}$ deconstructs a sequence Q into a subsequence Q' and the last element m. In Shacl-Lite, queues are sequences of messages and notifications and are processed right-to-left, meaning that the last message or notification in the sequence is the first to be processed. We denote both the empty set and the empty sequence using $\emptyset$. The notation $e_\Box[e]$ indicates that the expression e is part of a compound expression $e_\Box$, and should be reduced first before the compound expression can be reduced further.

Substitution Rules

$$
\begin{array}{rcl}
[v/x]x' & = & x' \\
[v/x]x & = & v \\
[v/x]e.f & = & ([v/x]e).f \\
[v/x]\text{null} & = & \text{null} \\
[v/x]e.m(\overline{e}) & = & [v/x]e.m([v/x]\overline{e})
\end{array}
\qquad
\begin{array}{rcl}
[v/x]m(\overline{x})\{e\} & = & m(\overline{x})\{e\} \text{ if } x \in \overline{x} \\
[v/x]m(\overline{x})\{e\} & = & m(\overline{x})\{[v/x]e\} \text{ if } x \notin \overline{x} \\
[v/x]e.f := e & = & ([v/x]e).f := [v/x]e \\
[v/x]e \leftarrow m(\overline{e}) & = & [v/x]e \leftarrow m([v/x]\overline{e})
\end{array}
$$

$$
\begin{array}{rcl}
[v/x]\text{let } x' = e \text{ in } e & = & \text{let } x' = [v/x]e \text{ in } [v/x]e \\
[v/x]\text{let } x = e \text{ in } e & = & \text{let } x = [v/x]e \text{ in } e \\
[v/x]\text{actor}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} & = & \text{actor}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \\
[v/x]\text{domain}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} & = & \text{domain}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \\
[v/x]\text{object}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} & = & \text{object}\{\overline{f := [v/x]e}, \overline{[v/x]m(\overline{x})\{e\}}\} \text{ if } x \neq \text{this} \\
[v/\text{this}]\text{object}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} & = & \text{object}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\}
\end{array}
$$

Figure 2: Substitution rules: x denotes a variable name or the pseudovariable this.

Any SHACL-LITE program represented by expression e is run using the initial configuration:

$$\mathcal{K}\langle\{\mathcal{A}\langle\iota_a, \emptyset, \{\iota_a\}, \emptyset, [\![e]\!]_{\iota_a}\rangle\}, \{\mathcal{D}\langle\iota_a, \text{W}, \emptyset\rangle\}, \emptyset\rangle$$

Actor-local reductions. Actors operate by perpetually taking the next message from their message queue, transforming the message into an appropriate expression to evaluate, and then evaluate (reduce) this expression to a value. When the expression is fully reduced, the next message is processed.

If no actor-local reduction rule is applicable to further reduce a reducible expression, this signifies an error in the program. The only valid state in which an actor cannot be further reduced is when its message queue is empty, and its current expression is fully reduced to a value. The actor then sits idle until it receives a new message.

We now summarize the actor-local reduction rules in Figure 4:

- LET: a "let"-expression simply substitutes the value of x for v in e.
- PROCESS-MESSAGE: this rule describes the processing of incoming asynchronous messages (not for notifications) directed at local objects. A new message can be processed only if two conditions are satisfied: the actor's queue Q is not empty, and its current expression cannot be reduced any further (the expression is a value v).
- INVOKE: a method invocation simply looks up the method m in the receiver object (belonging to some domain) and reduces the method body expression e with appropriate values for the parameters $\overline{x}$ and the pseudovariable `this`. It is *only* possible for an actor to invoke a method on an object within a domain on which that actor currently holds either a shared or exclusive view.
- FIELD-ACCESS, FIELD-UPDATE, FIELD-INITIALIZE: a field update modifies the owning domain's heap such that it contains an object with the same address but with an updated set of fields. Field accesses apply only to objects located in domains on which the actor has either an exclusive or shared view

Substitution Rules

$$
\begin{array}{rcl@{\qquad}rcl}
[\![x]\!]_{\iota_d} & = & x & [\![m(\overline{x})\{e\}]\!]_{\iota_d} & = & m(\overline{x})\{[\![e]\!]_{\iota_d}\} \\
[\![e.f]\!]_{\iota_d} & = & ([\![e]\!]_{\iota_d}).f & [\![e.f := e]\!]_{\iota_d} & = & ([\![e]\!]_{\iota_d}).f := [\![e]\!]_{\iota_d} \\
[\![\text{null}]\!]_{\iota_d} & = & \text{null} & [\![e \leftarrow m(\overline{e})]\!]_{\iota_d} & = & [\![e]\!]_{\iota_d} \leftarrow m([\![\overline{e}]\!]_{\iota_d}) \\
[\![e.m(\overline{e})]\!]_{\iota_d} & = & [\![e]\!]_{\iota_d}.m([\![\overline{e}]\!]_{\iota_d}) & & &
\end{array}
$$

$$
\begin{array}{rcl}
[\![\text{let } x = e \text{ in } e]\!]_{\iota_d} & = & \text{let } x = [\![e]\!]_{\iota_d} \text{ in } [\![e]\!]_{\iota_d} \\
[\![\text{object}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\}]\!]_{\iota_d} & = & \text{object}_{\iota_d}\{\overline{f := [\![e]\!]_{\iota_d}}, \overline{m(\overline{x})\{[\![e]\!]_{\iota_d}\}}\} \\
[\![\text{domain}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\}]\!]_{\iota_d} & = & \text{domain}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\} \\
[\![\text{actor}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\}]\!]_{\iota_d} & = & \text{actor}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\}
\end{array}
$$

Figure 3: Runtime domain substitution rules.

while field updates only applies in the case of an exclusive view. Initializing a field is done by reducing the runtime syntax, $f : e$ and has the same semantics as a field update. A field initialization will always be done right after the object's creation creation and does not require exclusive access to the owning domain.

- CONGRUENCE: this rule simply connects the actor local reduction rules to the global configuration reduction rules.

Rules for object, domain and actor literals. We summarize the creational reduction rules in figure 5:

- NEW-OBJECT: All object literals are tagged with the domain id of the lexically enclosed domain. The effect of evaluating an object literal expression is the addition of a new object to the heap of that domain. The fields of the new object are initialized to `null`. The literal expression reduces to a sequence of field initialize expressions. Note that we do not replace the `this` pseudovariable in the field initialize expressions. This means the reference to an object cannot leak before the object is completely initialized. The last expression in the reduced sequence is a domain reference r to the new object.

- NEW-DOMAIN: A domain literal will reduce to the construction of a new domain with a single object in its heap. Similarly to the rule for NEW-OBJECT, the domain literal reduces to a sequence of field initialize expressions. Initially, the domain's access modifier is set to F. After the field initialize expressions are reduced, a domain reference to the newly created object, $\iota_d.\iota_o$, is returned. $[\![e]\!]_{\iota_d}$ denotes a transformation that makes sure that all lexically nested object expressions are annotated with the domain id of the newly created domain. The substitution rules for this are straightforward and are left out of this paper.[1]

- NEW-ACTOR: when an actor ι_a reduces an actor literal expression, a new actor $\iota_{a'}$ is added to the set of actors of the configuration. A newly created domain is associated with that actor. The new domain's heap consists of a single new object ι_o whose fields and methods are described by the literal

[1]Full operational semantics can be found at `http://soft.vub.ac.be/~jdekoste/shacl/operational_semantics`

(LET)
$$\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\text{let } x = v \text{ in } e]\rangle \\ \to_a \mathcal{A}\langle \iota_a, S, E, Q, e_\Box[[v/x]e]\rangle$$

(PROCESS-MESSAGE)
$$\mathcal{A}\langle \iota_a, S, E, Q \cdot \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle, v\rangle \\ \to_a \mathcal{A}\langle \iota_a, S, E, Q, \iota_a.\iota_o.m(\overline{v})\rangle$$

(INVOKE)
$$\frac{r = \iota_d.\iota_o \qquad \iota_d \in S \cup E \qquad \mathcal{D}\langle \iota_d, l, O\rangle \in D \\ \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad \mathrm{m}(\overline{x})\{e\} \in M}{\mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[r.m(\overline{v})]\rangle\}, D, R\rangle \\ \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[[r/\text{this}][\overline{v}/\overline{x}]e]\rangle\}, D, R\rangle}$$

(FIELD-ACCESS)
$$\frac{\iota_d \in S \cup E \qquad \mathcal{D}\langle \iota_d, l, O\rangle \in D \\ \mathcal{O}\langle \iota_o, F, M\rangle \in O \qquad f : v \in F}{\mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\iota_d.\iota_o.f]\rangle\}, D, R\rangle \\ \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[v]\rangle\}, D, R\rangle}$$

(FIELD-UPDATE)
$$\frac{\iota_d \in E \\ o = \mathcal{O}\langle \iota_o, F \cup \{f : v'\}, M\rangle \\ o' = \mathcal{O}\langle \iota_o, F \cup \{f : v\}, M\rangle \\ D = D' \cup \{\mathcal{D}\langle \iota_d, \mathrm{w}, O \cup \{o\}\rangle\} \\ D'' = D' \cup \{\mathcal{D}\langle \iota_d, \mathrm{w}, O \cup \{o'\}\rangle\}}{\mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\iota_d.\iota_o.f := v]\rangle\}, D, R\rangle \\ \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[v]\rangle\}, D'', R\rangle}$$

(FIELD-INITIALIZE)
$$\frac{D = D' \cup \{\mathcal{D}\langle \iota_d, \mathrm{w}, O \cup \{\mathcal{O}\langle \iota_o, F \cup \{f : v'\}, M\rangle\}\rangle\} \\ D'' = D' \cup \{\mathcal{D}\langle \iota_d, \mathrm{w}, O \cup \{\mathcal{O}\langle \iota_o, F \cup \{f : v\}, M\rangle\}\rangle\}}{\mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\iota_d.\iota_o.f : v]\rangle\}, D, R\rangle \\ \to_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[v]\rangle\}, D'', R\rangle}$$

(CONGRUENCE)
$$\frac{a \to_a a'}{\mathcal{K}\langle A \cup \{a\}, D, R\rangle \\ \to_k \mathcal{K}\langle A \cup \{a'\}, D, R\rangle}$$

Figure 4: Actor-local reduction rules and congruence.

expression. The domain's access modifier is initialized to w and the domain is added to the set of exclusive domains of the newly created actor. That domain is "owned" by that actor, meaning that no other actors can ever acquire a view on that domain. The $[\![e]\!]_{\iota_d}$ syntax makes sure that all lexically nested object expressions are tagged with the domain id of the newly created domain. As in the rule for NEW-OBJECT, the object's fields are initialized to `null`. The new actor has an empty queue and will, as its first action, initialize the fields of the only object inside its associated domain. The actor literal expression itself reduces to a far reference to the new object, allowing the creating actor to communicate further with the newly spawned actor.

Asynchronous communication reductions. We summarize the asynchronous communication reduction rules in figure 6:

- LOCAL-ASYNCHRONOUS-SEND: an asynchronous message sent to a *local* object (i. e., an object owned by the same actor as the sender) simply appends a new message to the end of the actor's own message queue. The message send itself immediately reduces to `null`.
- DOMAIN-ASYNCHRONOUS-SEND: this rule describes the reduction of an asynchronous message send expression directed at a domain reference, i. e., a reference whose domain ι_d is not part of the set of

(NEW-OBJECT)

$$\frac{\begin{array}{c}\iota_o \text{ fresh} \qquad r = \iota_d.\iota_o \\ o = \mathcal{O}\langle \iota_o, \overline{f : \text{null}}, \overline{\text{m}(\overline{x})\{e'\}}\rangle\end{array}}{\begin{array}{l}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\text{object}_{\iota_d}\{\overline{f : e}, \overline{m(\overline{x})\{e'\}}\}]\rangle\}, D \uplus \{\mathcal{D}\langle \iota_d, l, O\rangle\}, R\rangle \\ \quad \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\overline{r.f : e}; r]\rangle\}, D \cup \{\mathcal{D}\langle \iota_d, l, O \cup \{o\}\rangle\}, R\rangle\end{array}}$$

(NEW-DOMAIN)

$$\frac{\begin{array}{c}\iota_o, \iota_d \text{ fresh} \qquad o = \mathcal{O}\langle \iota_o, \overline{f : \text{null}}, \overline{\text{m}(\overline{x})\{[\![e']\!]_{\iota_d}\}}\rangle \\ r = \iota_d.\iota_o \qquad d = \mathcal{D}\langle \iota_d, \text{F}, \{o\}\rangle\end{array}}{\begin{array}{l}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\text{domain}\{\overline{f : e}, \overline{m(\overline{x})\{e'\}}\}]\rangle\}, D, R\rangle \\ \quad \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\overline{r.f : [\![e]\!]_{\iota_d}}; r]\rangle\}, D \cup \{d\}, R\rangle\end{array}}$$

(NEW-ACTOR)

$$\frac{\begin{array}{c}\iota_{a'}, \iota_o \text{ fresh} \qquad r = \iota_{a'}.\iota_o \qquad o = \mathcal{O}\langle \iota_o, \overline{f : \text{null}}, \overline{\text{m}(\overline{x})\{[\![e]\!]_{\iota_{a'}}\}}\rangle \\ d = \mathcal{D}\langle \iota_{a'}, \text{W}, \{o\}\rangle \qquad a = \mathcal{A}\langle \iota_{a'}, \emptyset, \{\iota_{a'}\}, \emptyset, \overline{r.f : [r/\text{this}][\![e]\!]_{\iota_{a'}}}\rangle\end{array}}{\begin{array}{l}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\text{actor}\{\overline{f : e}, \overline{\text{m}(\overline{x})\{e\}}\}]\rangle\}, D, R\rangle \\ \quad \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[r]\rangle, a\}, D \cup \{d\}, R\rangle\end{array}}$$

Figure 5: Creational rules

ActorIds. Reducing an asynchronous message to a domain reference is semantically equivalent to reducing an exclusive view request on that reference and invoking the method synchronously while holding the view (See *View Reductions*). The domain reference is the target of the request and the body of the request is the invocation of the method on that reference. Further reduction of the `acquire` statement will eventually reduce the entire statement to `null`.

- FAR-ASYNCHRONOUS-SEND: this rule describes the reduction of an asynchronous message send expression directed at a far reference, i. e., a domain reference whose $\iota_{a'}$ is the same as another actor in the system. A new message is appended to the queue of the recipient actor $\iota_{a'}$ (top part of the rule). As in the LOCAL-ASYNCHRONOUS-SEND rule, the message send expression itself evaluates to `null`.

View reductions. We summarize the views reduction rules in Figure 7:

- ACQUIRE-VIEW: This rule describes the reduction of `acquire` expressions. This rule simply adds the view-request to the set of requests in the configuration. Note that this set is not an ordered set and thus requests can in principle be handled in any order. The acquire expression reduces to `null`.
- PROCESS-VIEW-REQUEST: Processing a view request removes that request from the set of requests and updates the access modifier of the domain. How the access modifier is allowed to transition from one value to another is described by the auxiliary function *lock*. Any request to a domain that is currently unavailable will not be matched by *acquire* and cannot be reduced as long as that domain remains unavailable. The auxiliary function *lock* yields the new value for the access modifier given the type of request and the current access modifier of the domain. As a result of processing a request a new notification is scheduled in the requesting actor's queue. Processing a view request can be done in parallel with reducing actor expressions.

(LOCAL-ASYNCHRONOUS-SEND)

$$\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\iota_a.\iota_o \leftarrow m(\overline{v})]\rangle$$
$$\rightarrow_a \mathcal{A}\langle \iota_a, S, E, \mathcal{M}\langle \iota_a.\iota_o, m, \overline{v}\rangle \cdot Q, e_\Box[\text{null}]\rangle$$

(DOMAIN-ASYNCHRONOUS-SEND)

$$\frac{\iota_d \notin \textbf{ActorId} \qquad r = \iota_d.\iota_o}{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[r \leftarrow m(\overline{v})]\rangle \rightarrow_a \mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\text{acquire}_{\text{E}}(r)\{r.m(\overline{v})\}]\rangle}$$

(FAR-ASYNCHRONOUS-SEND)

$$\frac{A = A' \uplus \{\mathcal{A}\langle \iota_{a'}, S', E', Q', e'\rangle\} \qquad A'' = A' \cup \{\mathcal{A}\langle \iota_{a'}, S', E', \mathcal{M}\langle \iota_{a'}.\iota_o, m, \overline{v}\rangle \cdot Q', e'\rangle\}}{\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\iota_{a'}.\iota_o \leftarrow m(\overline{v})]\rangle\}, D, R\rangle \rightarrow_k \mathcal{K}\langle A'' \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\text{null}]\rangle\}, D, R\rangle}$$

Figure 6: Asynchronous message rules

- PROCESS-VIEW-NOTIFICATION: Processing a notification will add the domain id, ι_d, to the set of shared or exclusively accessible domains in the actor. Analogous to the processing of messages, a new notification can be processed only if two conditions are satisfied: the actor's queue Q is not empty, and its current expression cannot be reduced any further (the expression is a value v). The actor's set of available shared or exclusive domains is updated according to the request using the *add* function. Processing a notification changes the expression that the actor is currently evaluating to the expression in the notification, followed by a release expression.
- RELEASE-VIEW: Releasing a view on the domain removes that domain id from the set of shared or exclusively accessible domains of the actor. The access modifier of the domain is also updated, potentially allowing other view requests on that domain to be processed. The release statement, which is always the last statement that is reduced by an actor before reducing other messages in its queue, also reduces to `null`.

Auxiliary functions and predicates. The auxiliary function $unlock(t, l)$ describes the transition of the value of an access modifier when releasing it. If a domain was locked for exclusive access its lock will be W (write) and can be changed to F (free). If that resource was locked for shared access we transition either to F or subtract one from the read modifier's value. Similar to the *unlock* rule, the $lock(t, l)$ rule describes the transition of the value of the access modifier of a shared resource when acquiring it. These two rules effectively mimic multiple-reader, single-writer locking.

The *add* and *subtract* rules with four parameters describe the updates to the set of shared, S, and exclusive, E, domain ids of an actor. The *add* rule adds domain ids to both sets while *subtract* rule subtracts domain ids from both sets.

1.4. Object creation in SHACL

In the operational semantics an object is created by initializing its fields to `null` and then reducing a number of field initialization expressions. Object expressions are always lexically nested within an actor or domain and as such, an object can only be created by an actor when that actor has shared or exclusive access to that domain. The reason we do not use field updates for initializing an object is that field updates require the actor to have an exclusive view on the domain of the object. Meaning that, when using field updates, object creation would not be possible in shared mode. This would severely limit the expressiveness of our model. Field initialization expressions of objects are not allowed to use the `this` pseudovariable to refer to the object that is being created. This restriction ensures that object references to an object under construction cannot be leaked to other actors before the object is completely initialized. This restriction is necessary to avoid data races, which can otherwise occur when an object is created inside a domain that was acquired in shared mode. If such an object could be shared with other actors before it is fully initialized,

(ACQUIRE-VIEW)

$$\frac{\iota_d \notin \mathbf{ActorId}}{\begin{array}{c}\mathcal{K}\langle A \uplus \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\text{acquire}_t(\iota_d.\iota_o)\{e\}]\rangle\}, D, R\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\text{null}]\rangle\}, D, R \cup \{\mathcal{R}\langle \iota_a, \iota_d, t, e\rangle\}\rangle\end{array}}$$

(PROCESS-VIEW-REQUEST)

$$\frac{l' = lock(t, l) \qquad D = D' \uplus \{\mathcal{D}\langle \iota_d, l, O\rangle\} \qquad D'' = D' \uplus \{\mathcal{D}\langle \iota_d, l', O\rangle\}}{\begin{array}{c}\mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e\rangle\}, D, R \uplus \{\mathcal{R}\langle \iota_a, \iota_d, t, e\rangle\}\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, \mathcal{N}\langle \iota_d, t, e\rangle \cdot Q, e\rangle\}, D'', R\rangle\end{array}}$$

(PROCESS-VIEW-NOTIFICATION)

$$\frac{S', E' = add(\iota_d, t, S, E)}{\begin{array}{c}\mathcal{A}\langle \iota_a, S, E, Q \cdot \mathcal{N}\langle \iota_d, t, e\rangle, v\rangle \\ \rightarrow_a \mathcal{A}\langle \iota_a, S', E', Q, e; \text{release}_t(\iota_d)\rangle\end{array}}$$

(RELEASE-VIEW)

$$\frac{\iota_d \notin \mathbf{ActorId} \qquad l' = unlock(t, l) \qquad S', E' = subtract(\iota_d, t, S, E) \qquad D = D' \uplus \{\mathcal{D}\langle \iota_d, l, O\rangle\} \qquad D'' = D' \cup \{\mathcal{D}\langle \iota_d, l', O\rangle\}}{\begin{array}{c}\mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S, E, Q, e_\Box[\text{release}_t(\iota_d)]\rangle\}, D, R\rangle \\ \rightarrow_k \mathcal{K}\langle A \cup \{\mathcal{A}\langle \iota_a, S', E', Q, e_\Box[\text{null}]\rangle\}, D'', R\rangle\end{array}}$$

Figure 7: Views reduction rules.

other actors may non-deterministically observe updates to the object as it is being initialized by its creating actor.

In mainstream OO languages such as C# or Java, constructors are allowed to refer to `this`. It is a known problem that this can lead to similar issues, as other objects may then start to interact with objects that are only half-initialized [4].

2. Acknowledgements

Joeri De Koster is supported by a doctoral scholarship granted by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

Tom Van Cutsem is a Postdoctoral Fellow of the Research Foundation, Flanders (FWO)

References

[1] T. V. Cutsem, C. Scholliers, D. Harnie, An operational semantics of event loop concurrency in ambienttalk, Tech. rep., Vrije Universiteit Brussel (2012).

[2] J. Schäfer, A. Poetzsch-Heffter, Jcobox: Generalizing active objects to concurrent components, ECOOP 2010–Object-Oriented Programming (2010) 275–299.

[3] M. Felleisen, R. Hieb, The revised report on the syntactic theories of sequential control and state, Theoretical computer science 103 (2) (1992) 235–271.

[4] J. Gil, T. Shragai, Are we ready for a safer construction environment?, in: S. Drossopoulou (Ed.), ECOOP 2009 Object-Oriented Programming, Vol. 5653 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 495–519.

Auxiliary functions and predicates

$$
\begin{array}{rcl@{\qquad}rcl}
lock(\textsc{e},\ \textsc{f}) & \stackrel{def}{=} & \textsc{w} & unlock(\textsc{e},\ \textsc{w}) & \stackrel{def}{=} & \textsc{f} \\
lock(\textsc{s},\ \textsc{f}) & \stackrel{def}{=} & \textsc{r}(1) & unlock(\textsc{s},\ \textsc{r}(1)) & \stackrel{def}{=} & \textsc{f} \\
lock(\textsc{s},\ \textsc{r}(i)) & \stackrel{def}{=} & \textsc{r}(i+1) & unlock(\textsc{s},\ \textsc{r}(i)) & \stackrel{def}{=} & \textsc{r}(i-1) \\
\\
add(\iota_d, \textsc{e}, S, E) & \stackrel{def}{=} & S, E \cup \{\iota_d\} & subtract(\iota_d, \textsc{e}, S, E \uplus \iota_d) & \stackrel{def}{=} & S, E \\
add(\iota_d, \textsc{s}, S, E) & \stackrel{def}{=} & S \cup \{\iota_d\}, E & subtract(\iota_d, \textsc{s}, S \uplus \iota_d, E) & \stackrel{def}{=} & S, E
\end{array}
$$