Sepro - System for modelling and simulation of qualitative network dynamics

Stefan Urbanek <stefan.urbanek@gmail.com>

2018-08-10

Inspired by biochemistry.

Introduction

Objective of this project is to test alternative approach for simulation of network problems, examine possibility of purely qualitative approach, find primitives of non-conventional computation of network problem solving and develop a simulator prototype and set of models that demonstrate the system.

System Design Principles

Development of a system for modeling network complexity is non-trivial as assumptions based on our more sophisticated knowledge might start creeping into the design process. Such assumption creep can corrupt the system and therefore the outcome of the models. As we don't foresee yet how the ultimate design looks like, we will use evolutionary iterative approach to the design process. To stay on track, we constrain our evolutionary process by design principles:

- *Completeness and clarity of model description.* The model described by the system has to be complete and should not require other information than the system specification to be understood.
- *Minimal set of assumptions.* Assumptions for behaviour or structure should be kept to minimum. System should provide only primitives and basic mechanisms from which more complex behavior or structure is to be composed. The primitives should be as simple as possible. New features should be evaluated carefully whether they can't be implemented using existing mechanisms. If they can, they should be omitted.
- *No explicit control flow.* There should be no mechanisms in the system that would guarantee model creators control flow (evaluation order) in

atomic way. Model-specific order should always be vulnerable for potential interference from another model that might be composed.

- *Iterative simulation.* Modeled system's state changes iteratively through state transitions. Absolute time (number of steps) is not observable to the model.
- *Parallel.* All components of the model are assumed to operate in parallel fashion even though the system's implementation might be fully or partially serial. Effects of serial computation to the simulation result is considered an inevitable error of the concrete implementation of the system. Model results should carry the information about the nature of the computation engine.

Model

The modeled universe comprises of a system's state and description of the system's dynamics. The state is a graph where we call nodes *objects* which have qualitative properties associated with them. The system's dynamics is a collection of rules describing system's behavior called *actuators*.

To be able to refer to particular components of the model in a human readable form we use *symbols*. Each symbol can refer to either particular component of the model or a type of component.

Definition Model is a tuple $M := (S, S \to t, A, G)$ where S is a set of symbols, $S \to t$ is a symbol type table where t is a symbol type, A is a collection of actuators and G is a graph representing system's state.

Definition Symbol table $S \to t$ is a mapping between a symbol and it's type:

$$T := s \to t | s \in S, t = \begin{cases} \text{tag} \\ \text{slot} \\ \text{actuator} \end{cases}$$

We say that a symbol s^t is of type t when $s^t \in S \wedge T(s^t) = t$. Set of symbols S^t is a set where each symbol is of type t.

Object is indivisible entity representing an instance of relevant concept within simulated universe. It is a carrier of qualitative properties - *tags*. State of an object is denoted by a set of tags.

Definition Object graph G is labeled oriented multigraph of objects (edges) and relationships (vertices) (O, R). Relationship is a tuple $\{s^{\text{slot}}, o\}$ where $s^{\text{slot}} \in S$ and $o \in O$

Definition Object's qualitative state is a set of symbols $\{s_1^t, s_2^t, ..., s_n^t\} | s_i^t \in S^{\text{tag}}$. We will write it as tags(o)

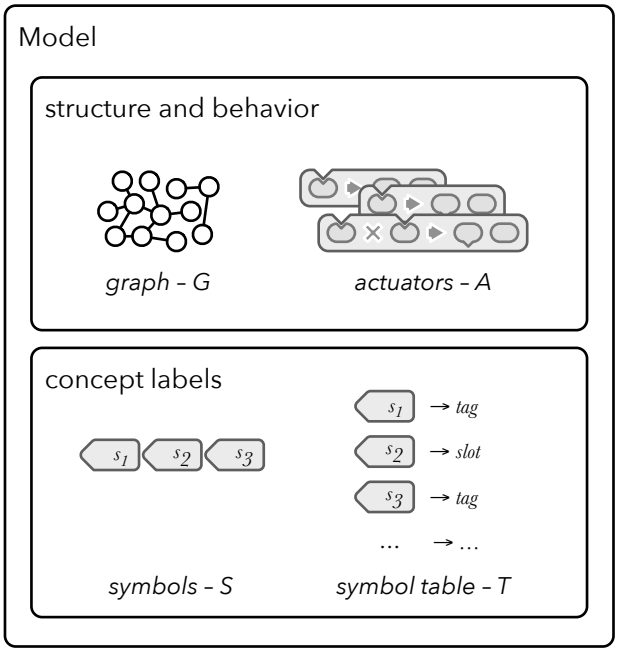

Figure 1: Model

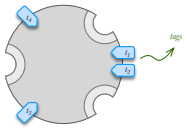

Figure 2: Object tags

Slot is a relational property of an objects that references other objects. Slot is a label of an edge of the object graph. We will use the letter s to denote a slot.

Proposition We say that object o has slots $\{s_1^s, s_2^s, ..., s_n^s\}$ if there exist edges $\{o, s_i^s\}$. We will write it as slots(o)

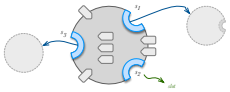

Figure 3: Object slots

Before we proceed with the system's dynamics, we need to define one more design concept of the system: *local context*.

Definition *Local context* of an object o is a subgraph $G^L \subseteq G$ with all objects and relationships where object o is the inital vertex.

In other words, local conetxt of an object o is a subgraph within a graph distance of 1 from the object o.

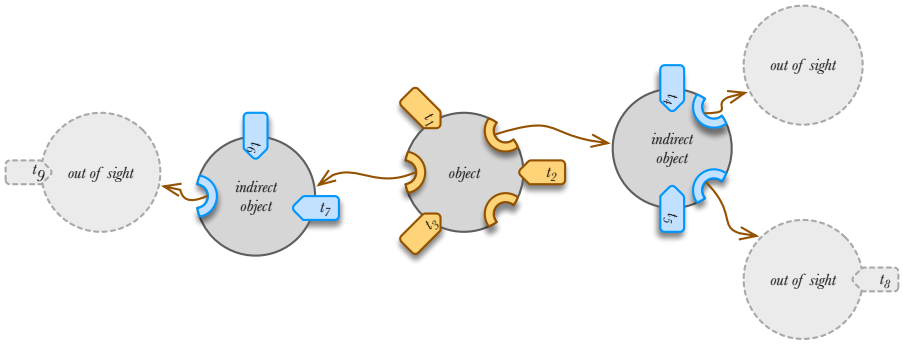

Figure 4: Local context

Before we move on with defition of other concepts, we need to define how objects in the graph can be reffered to. We need to be able to refer to an object within the local context relatively to the object being considered for evalutaion.

Definition *Subject mode* is a relative reference to an object in the graph given initial object.

$$m := \begin{cases} \text{direct} \\ \text{indirect}(s^{\text{slot}}) \end{cases}$$

Subject mode is used to determine which object will be used for pattern matching in the selector or for applying state transitions. We call the object that is to be used for evaluation *effective subject*. *Direct* subject mode means that the

effective subject is the object being evaluated is to be considered. *Indirect* subject mode means that the *effective subject* is the object which is terminal vertex o_t of the relationship $\{s^{\text{slot}}, o_t\}$ where the initial vertex is the object being evaluated.

Model Dynamics

The main concepts for the model dynamics are:

- *Actuator* is a description of an atomic state transition of objects within the graph matching a pattern.
- *Selector* is a match pattern or objects subject to transition
- *Transition/Modifier* is a description of state change affects either object's state or local relationships

Pattern matching or state transitions can happen only within a *local context*. The *local context* is an intentional design limitation which restricts that object state and graph transitions can happen only within distance of 1 from the selected object. Assumption here is that if we want to reach an object with larger distance we have to use multiple steps and therefore be open for interference - which is desired by design.

Actuators

Actuators can be thought as declarations of graph rewrite rules combined with object state transitions. They are applied to the whole graph[1]. We assume that all actuators operate on all objects at once.[2]

The graph can be modified either through a state of a single object or a state of two object in their hypothetical interaction.

For observation and controlled simulation purposes the actuators have also a *control signallig* associated with them. We will discuss the control mechanism later.

Unary Actuator describes transition of object's state and local relationships based on previous object's state. Only object that matches a pattern or it's direct neighbours can be affected.

Binary Actuator is a conditioned transition of an object as a result of cartesian product of two objects matching two selector patterns. Either of two objects can undergo transition based on state of any of the objects in the cartesian product tuple.

[1] Here we assume only lowest level of the Sepro model without a constraints level. Higher levels as well as constraints are out of scope of this article.

[2] This is idealized assumption which has technical implementation limitations that we will discuss later.

Figure 5: Unary Actuator

Definition *Unary actuator* is a tuple $(\sigma, m \to T^1, n)$ where σ is a selector, m is subject mode and T^1 is unary transition. n is a control signal (as in "notification").

Figure 6: Binary Actuator

Binary actuator is the only way how a new connections to potentially unrelated objects (no direct reference) might happen.

Definition *Binary actuator* is a tuple $(\sigma_l, \sigma_r, m_l \to T_l^2, m_r \to T_r^2, \Gamma)$ where σ_l and σ_r are left and right selector respectively, T_l^2 and T_r^2 are left and right binary transitions on effective subject specified by left subject mode m_l and right subject mode m_r respectively. n is a control signal.

We would use the term *hand* to refer to the left or right selector or transition.

Model language declaration of a binary actuator is:

```
REACT actuator_name
    WHERE (selector_patterns)
        ON (selector_patterns)
        DO binary_transitions
```

Selector

Selector is a pattern description that matches properties of objects and their local context. Pattern is a collection of multiple predicates that test qualitative properties or existence of relationships of an object. An object matches a selector pattern when:

- a direct predicate matches the object
- an indirect predicate matches object's direct neighbors

The predicates can tets for:

- Tags associated with an object: true if selector's tags $\subset$ object's tags
- Tags not associated with an object: true if object's tags$\cap$selector's tags $= \emptyset$
- Graph contains an edge from a specific slot
- Graph does not contain an edge from a specific slot

Definition *Selector* is a patter description

$$\sigma := \begin{cases} \text{all} \\ \text{match}(m \to \Pi) \end{cases}$$

where m is a subject mode and Π is a selector pattern. We say that object matches a selector when the selector is all or when the *effective subjects* of the object match all the selector patterns Π.

Definition *Symbol presence* p is a case

$$p := \begin{cases} \text{present} \\ \text{absent} \end{cases}$$

.

Definition *Selector pattern* Π is a tuple of mappings $(S^t \to p, S^s \to p)$ where p is symbol's presence. An object matches the selector pattern if all of the following are true:

$$\begin{aligned} & \{s^t | s^t \to \text{present}\} \subset \text{tags}(o) \\ \wedge\ & \text{tags}(o) \cap \{s^t | s^t \to \text{absent}\} = \emptyset \\ \wedge\ & \{s^s | s^s \to \text{present}\} \subset \text{slots}(o) \\ \wedge\ & \text{slots}(o) \cap \{s^s | s^s \to \text{absent}\} = \emptyset \end{aligned}$$

The language representation of the selector pattern is either a word `ALL` or a list of symbols. Assume we have symbols `open`, `empty` referring to tags and symbol `next` referring to a slot. For example the selector in the following actuator matches all objects that have tag `open` set, have no `empty` tag set and there exists a relationship at slot `next` from the object:

```
WHERE (open, !empty, next) ...
```

All symbols are considered to be in direct subject mode by default. Indirect subject mode in the selector can be represented by object qualifier "dot" operator as `indirection.symbol`[3]. For example:

```
WHERE next.open ...
```

The above matches an object where an object referred through slot `next` has a tag `open` set.

State Transitions

State transitions (further just *transitions*) are descriptions of qualitative changes of the object graph. They operate on objects and their neighbors within their

[3]Unlike in common general purpose programming languages, the indirection can not be chained as in `deep.deeper.deepest.symbol` due to the *local context* constraint.

local context. Proposed transitions are non-divisable primitives we assume being sufficient for any desired graph state transformations when composition of the transitions is used.

The concrete object that is subject to transition is called *effective subject* of the transition and is determined by the subject mode in the actuator.

There are two kinds of transitions: qualitative state of an object and qualitative state of the graph. The first one operates on object's qualitative properties - *tags* and the later operates on graph's relationships. The tags can be associated or disassociated from an object. The relationships can be *bound* and *unbound* within the local context of the effective subject.

Unary Transition

Definition *Unary transition* is a tuple $T^1 = (S^{\text{tag}} \to p, S^{\text{slot}} \to \mu^1)$ where the first element is a qualitative transition of the effective subject and the second element is a graph edge change from the effective subject to effective target as described by the unary target specifier μ^1.

If the $p =$ present then the S^{tag} is associated with the effective subject. If the $p =$ absent then the S^{tag} is dissociated with the effective subject. [4]

Definition *Unary target specifier* μ^1 is a case:

$$\mu^1 := \begin{cases} \text{unbind} \\ \text{subject} \\ \text{in\_subject}(S^{\text{slot}}) \\ \text{indirect}(S^{\text{slot}}, S^{\text{slot}}) \end{cases}$$

The unbind case specifies that the edge from the effective subject is to be removed. subject denotes that the target is the effective subject itself, therefore creating a self-loop. Effective target of the in_subject case is the object referred by the specified slot from the effective subject. The indirect effective target is an object reffered to by the path of two slots from the effective subject.

The above gives us the following potential subject mode combinations for creating an edge using unary actuator. Let's assume the effective subject having slots s and t, and the object referred to by slot s having slot i, object referred to by slot t having slot w.

[4] Alternative and more readable or understandable way of specifying which tags are to be associated or disassociated with an object would be to use two sets of tags: *set* and *unset*. However if the intersection of the sets is non-empty, the behaviour would be undefined. Using the mapping we prevent such situation from happening by design.

Effective subject	Effective target	Edge
direct	none	*removed*
direct	subject	$s \to$ self
direct	in_subject(t)	$s \to t$
direct	indirect(t, w)	$s \to t.w$
indirect(i)	none	*removed*
indirect(i)	subject	$s.i \to$ self
indirect(i)	in_subject(t)	$s.i \to t$
indirect(i)	indirect(t, w)	*not atomic*

Constraint Indirection of effective subject and effective target is not permitted, as the operation can be achieved by by composing two separate actuators: one for pulling indirect object closer to the effective subject and second for performing indirect bind to the pulled-in subject and unbinding the subject.

Binary Transition

Binary transition is analogous to the unary transition with one difference: the effective target specifier can specify one of the two "hands" of the selector.

Effective subject of the binary transition is the subject selected by corresponding hand selector. For the left hand selector σ_l the corresponding transition is $T^2 l$ and the effective subject of the transition is the subject determined by σ_l. Analogously for the right hand transition the effective subject is determined by the σ_r.

Transition hand can affect only qualities of the effective subject on the same hand similarly to unary transition. Although transition hand can have effective target from the same hand or from the other hand. This allows us to create new relationships between objects that are from disconnected parts of the graph. We refer to the effective subject from the other selector simply as other.

Definition *Binary transition* is a tuple $T^1 = (S^{\text{tag}} \to p, S^{\text{slot}} \to \mu^2)$ where the first element is a qualitative transition of the effective subject and the second element is a graph edge change from the effective subject to effective target as described by the binary target specifier μ^2.

The first element of the tuple for tags is the same as the mapping in the unary transition.

Definition *Binary target specifier* μ^2 is a case:

$$\mu^1 := \begin{cases} \text{unbind} \\ \text{other} \\ \text{in\_other}(S^{\text{slot}}) \end{cases}$$

The unbind case specifies that the edge from the effective subject is to be removed. other denotes that the target is the effective subject of the other hand. Effective target of the in_other case is the object referred by the specified slot from the other hand's effective subject.

Note that there is no indirection in the binary transition as it can be achieved by composing multiple transitions. Neither there is possibility to create binding within the same effective subject as it can be achieved by composing as well [5].

The following table lists allsubject mode combinations for creating an edge between objects in the binary actuator. Let's assume the effective subject on one hand having slots s and i, and the effective subject on the other hand having slot t.

Effective subject	Effective target	Edge
direct	none	*removed*
direct	other	$s \rightarrow$ other
direct	in_other(t)	$s \rightarrow$ other.t
indirect(i)	none	*removed*
indirect(i)	other	$s.i \rightarrow$ other
indirect(i)	in_other(t)	$s.i \rightarrow$ other.t

Transition Modes Summary

The following figure shows all possible graph transitions for edge creation for both unary and binary actuators:

Binary modifier have limited ability to modify the state by design. Unbinding in a binary modifier can be achieved by a combination of binary state change and an unary `UNBIND` modifier. Indirection in the binary modifier can be achieved by a combination of binary direct subject state change and an unary modifier.

Note that all state changes beyond distance of 1 from the selected object must be composed of multiple transitions that propagate through the network. Susceptibility to being affected by other actuators along the way is intended design feature.

Simulation

The simulation is virtually indefinite iterative evaluation of the model's actuators that operate on system's state.

[5] This is a design decision that we are proposing at this stage of system's evolution. We have no firm opinion whether bindings within the same hand should be allowed or not at this moment.

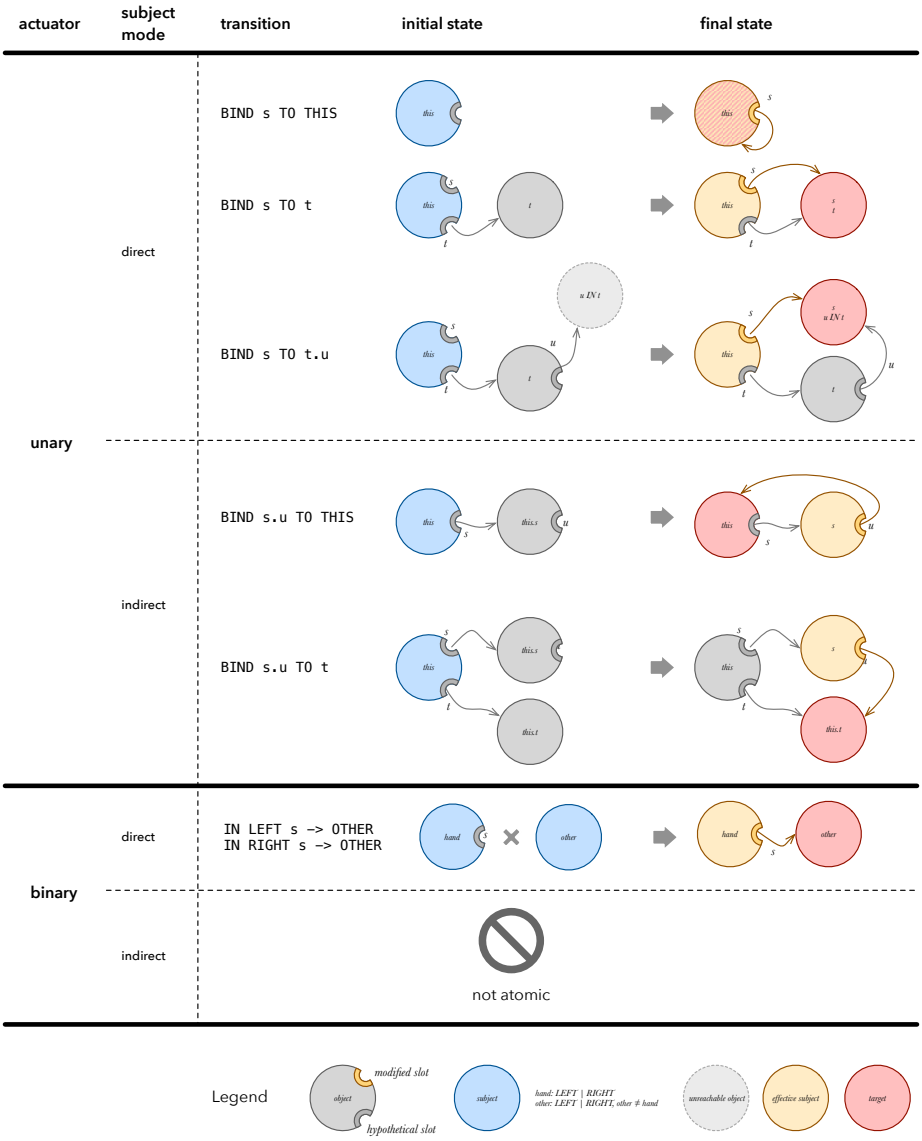

Figure 7: Possible graph transitions for edge creation

Given inspiration in biochemistry, the nature of the Sepro system does not impose any evaluation order of the actuators and selection order of objects. However, to be able to perform the simulation on Von-Neumann computer architecture which is sequential, we need to define the order of events in the simulation, and understand it's impact on the simulation result. The emphasis is more on the explicit impact description than on the actual order definition. The evaluation order is a meta-problem that we are not trying to solve yet, but we need to propose few solution to start with.

Assumption In this evolutionary step of the system we consider the time to be unified globally. That means that time is the same for every entity of the system.

Having global time of discrete nature, we can refer to each state by global time reference t and can say that state of the system at time t is described as a state of objects and bindings at time t.

Definition Simlation step Δ is an approximation of system's transition in form of a function

$$G^{t+1} = \Delta(G^t, M)$$

where G^0 is the graph G from the model M.

We have to keep in mind that the Δ is not a true simulation mechanism, just an approximation. It is so due to the assumption of global time and potential effects of linearization.

During the simulation step the following happens:

- Every unary actuator is tested against each object and the associated unary transition is applied to the objects that match the actuator's selector.
- For every binary actuator a cartesian product of objects matching left selector and right selector is determined and binary transitions are applied to their respective effective subjects.
- Control signals are gathered and provided to the simulation controller or observer.

As mentioned above, the order how the actuators, their evaluation and application is executed is left to the concrete implementation of the simulation engine. The decision whether the simulation is performed in parallel and what kind of parallelism is used is an implementation choice of the simulation engine. It has to be remembered that, as mentioned before, the simulation might be highly sensitive to the order of execution.

"Sequential Scan" Simulation Method

Here we propose a simple, quite primitive yet straightforward simulation method: *Sequential scans with actuators first.* This method performs the *simulation step* in a single thread and considers the time to be system-global.

The simulation step can be described in a pseudo-language as:

```
FOR actuator IN unary actuators DO:
    evaluate unary actuator

FOR actuator IN binary actuators DO:
    evaluate binary actuator
```

We serialise the simulation process by applying the transitions of the system in the order as given by a lazy selection algorithm described below. The method is analogous to vertical and horizontal line scan of a CRT screen where an object can be seen as a point on the screen and where the beam traverses the points in fixed pattern. One simulation step can be modelled by a single full scan of the whole screen. Once the beam touches a point on the screen, it does not go back within the same full screen scan.

The unary actuator scan is depicted in the following figure:

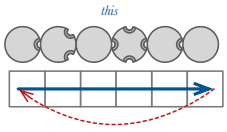

Figure 8: Unary scan

The evaluation of unary actuator is a single pass through the lazy selection of objects matching actuator's predicates:

```
selection := objects matching actuator selector

FOR object IN selection DO:
    IF object matches selector:
        apply actuator transitions to object
```

The binary actuator "scans" a cartesian product of the "left" and "right" selector of the actuator:

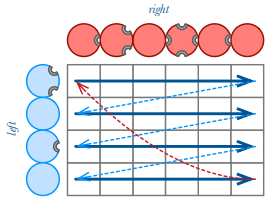

Figure 9: Binary actuator - scan of cartesian product

The evaluation of the binary selector is as follows:

```
selection L := GET objects matching left selector of actuator
selection R := GET objects matching right selector of actuator
```

```
FOR left IN L DO:
    FOR right IN R DO:
        IF left does not match left selector:
            CONTINUE
        IF right does not match right selector:
            CONTINUE

        apply transition to left and right
```

The inner conditions are to filter out objects that might have been already modified in the scan pass and might not fit the selection predicates any more.

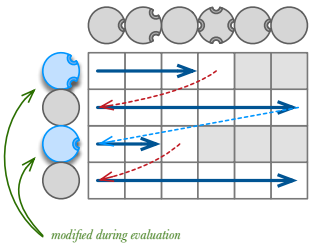

Figure 10: Binary actuator - skipping

Known Issues

The scan method described above has two major factors that influence the the simulation's outcome:

- Order in which actuators are evaluated.
- Order in which the objects are provided to the filter.

Let's consider two actuators A and B evaluated in the same order: first A then B. If an object does not match predicates of A, matches predicate for B and actuator B modifies the object in a way that it would match actuator A, the object is not evaluated again with the actuator A. We will call this *actuator order error*.

Let's consider order of objects $o_1, o_2, \ldots, o_n$ and an actuator that by evaluating o_1 modifies o_2 in a way that o_3 will not match the actuator's predicate (will lose candidacy, will not be visited). If we provide another order, for example reverse order, then o_2 will be visited. We will call this *object selection order error*.

Here we suggest that how the simulation sub-steps are ordered must be known fact to the simulator user.

For further research, it might be interesting to further investigate effect of ordering to the outcome of certain models. For example:

- Randomization of the actutors, including unary and binary.
- Randomization of the objects for each iteration and actuator.
- Object-first pass: the outer loop is object loop, the inner loops are actuator loops.

In a potential virtual laboratory where one might test different kinds of ordering the controller might provide mechanisms to compare different outcomes based on the orderings. This investigation is out of scope of this article, left as an exercise to the reader.

Parallel Evaluation

Massively parallel evaluation is the ultimate goal of the system as it closely mimics the behaviour in the real world. By *massively parralel* we mean one processing unit per object per actuator observing the relevant context of the unit-associateid object.

Simplified parallelism can be achieved by splitting the object graph into smaller parts, performing partial evalutaions and then consolidating the results. This is a whole are to be explored as it opens many questions, such as:

- How to synchronise the simulation states?
- How to resolve potential modifier-predicate order conflicts?
- How to consolidate conflicting modifications?
- How the consolidation method affects the outcome of the simulation?
- Which parameters of the simulation configuration affect potential errors of the simulation and in which way?

Control Signaling

The language gives a possibility to provide signals to the simulator. Signals are triggered together with activation of the associated actuator. The control signalling has no direct effect on the model and it's interpretation is given by the simulator. It can be thought as an action to communicate unidirectionally with the observer.

There are three kinds of signals: *notify*, *trap* and *halt*. The *notify* and *trap* signals can carry a set of tags associated with them to provide more information to the signal handlers.

- `NOTIFY` signals to the simulator and expectes no interruption of the simulation. State of the simulation must not be changed by the handler. Use case: monitor reached goals; trigger/start/stop measurement; visualize a state of interest
- `TRAP` signals and interrupts the simulator, giving it possibility to resume the simulation after the signal has been handled. State of the simulation might

be changed by the handler. Use case: goal reached and user interaction is expected; a product has been created

- `HALT` signals to the simulator to terminate the simulation without possibility of resuming it. Use case: invalid state has been reached, resuming the simulation might yield non-sensical results; final state has been reached and resuming the simulation might affect the result in non-meaningful way.

Language

For the Sepro system we propose a domain specific modeling language. The language convers three aspects of the model: object prototypes, initial graph structures and simlator or observer related information.

Model and Model Objects

The model is a list of model objects: symbol definitions, actuators and structures.

model = { *model_object* }

model_object = *symbol_definition* | *unary_actuator* | *binary_actuator* | *structure*

Symbols

The *symbol* is an identifier that can contain letters, decimal digits or the underscore `_` character. It can not start with a digit. For example `open`, `next`, `site_a`, `site_b`.

Each symbol in the model represents an instance of a type. There can be only one type associated with a symbol within the model. Type of a symbol can be specified explicitly or is determined by the compiler from the first use of the symbol in the model. Use of a symbol for different types results in an error. For example if a symbol is used as a tag it can not be used to label a relationship between objects.

Examples of explicity symbol definitions:

```
DEF TAG open
DEF TAG closed
DEF SLOT next
DEF SLOT site_a
```

Grammar:

symbol_definition = `"DEF"` *symbol_type symbol* *symbol_type* = `"SLOT"` | `"TAG"` | `"STRUCT"` | `"ACTUATOR"`

When an indirect symbol needs to be specified we use the . (dot) symbol qualification:

qualified_symbol = [*symbol* `"."`] *symbol*

Structures

Structure is a definition of a group of objects - a subgraph, that can be used to initialize the world. Structure contains list of objects and bindings (edges).

struct = `"STRUCT"` *symbol* { *struct_item* }

struct_item = *object* | *binding*

Structure objects have identifiers that are valid withing the scope of the structure. The identifiers are used to refer to objects in the structure binding specification.

object = `"OBJ"` *symbol* `"("` { *symbol* } `")"`

The bindings within structure can refer to objects within the same structure:

binding = `"BIND"` *symbol* `"."` *symbol* `"TO"` *symbol*

Example:

```
STRUCT triangle
    OBJ a (node)
    OBJ b (node)
    OBJ c (node)
    BIND a.next -> b
    BIND b.next -> c
    BIND c.next -> a
```

Worlds

World is a container specifying initial state of the simulation. It can be thought as a list of "ingredients of the simulation primordial soup".

In the language more worlds can be specified in the model, despite only one world being used as a starting state of the system. If more worlds are specified then the one with name `main` is used if not specified explicitly otherwise.

world = `"WORLD"` *symbol* { *world_item* }

world_item = *integer symbol*

For example the following world will yield 10 copies of structure *jar* and 10 objects (out of structure) with tag *lid*:

```
WORLD main
    10 jar
    10 (lid)
```

Actuators

selector = `"ALL"` | `"("` { *symbol_presence* } `")"`

symbol_presence = [`"!"`] *qualified_symbol*

unary_actuator = `"ACT"` *symbol* `"WHERE"` *selector* { *unary_transition* }

unary_transition = `"IN"` *unary_subject* { *modifier* }

unary_subject = `"THIS"` [`"."` *symbol*] | *symbol*

modifier = `"BIND"` *symbol* `"TO"` *qualified_symbol* | `"UNBIND"` *symbol* | `"SET"` *symbol* | `"UNSET"` _symbol

binary_actuator = `"REACT"` *symbol* `"WHERE"` *selector* `"ON"` *selector* { *binary_transition* }

binary_transition = `"IN"` *binary_subject* { *modifier* }

binary_subject = (`"LEFT"` | `"RIGHT"`) [`"."` *symbol*]

Example: Linker

We made a simple example for the purpose of demonstration of the language that we call 'Linker'. The model has two kinds of objects: a `link` node and a 'catalyst' we call `linker`. The linker has two sites: `site_a` and `site_b`. The objective is to create chains of links when we put the linker and couple of links into the simulation container.

The proposed process is:

- bind one site of the linker to a free link
- bind another site of the linker to another free link
- when both sites are bound then bind the links together
- release one link and free the site
- continue from binding another link to the free site

The program listing below contains actuators that satisfy the process above. One might have noticed that the example contains explicit order specified by transition of the linker through

```
DEF SLOT site_a
DEF SLOT site_b
DEF SLOT next
DEF TAG linker
DEF TAG link

REACT primer
    WHERE (linker !site_a)
    ON (free link)
    IN LEFT
```

```
        BIND site_a TO other
        SET wait_right

    IN RIGHT
        UNSET Free

REACT _wait_right
    WHERE (wait_right)
    ON (free link)
    IN left
        BIND site_b TO other
        UNSET wait_right
        SET chain
    IN right
        UNSET free

ACT _chain
    WHERE (chain)
    IN this
        UNSET chain
        SET advance
    IN this.site_a
        BIND next TO site_b

ACT _advance
    WHERE (advance)
    IN this
        BIND site_a TO site_b
        UNSET advance
        SET cleanup

ACT _cleanup
    WHERE (cleanup)
    IN this
        BIND site_b TO none
        UNSET cleanup
        SET wait_right

WORLD main
    30 (free link)
    3 (linker)
```

Rejected Ideas

The ideas listed here were either rejected or postponed for further re-consideration or re-design.

Root Object

The *root object* served as a globally referencable state. From simulation dynamics perspective it was no different to any other object. The only difference with

other objects was that the root object could be referenced explicitly by a symbol `ROOT`. Objects were able to react with the root object through binary selectors where one of the selector operands was a root object.

From the original proposal:

> There might be situations where we need to consider a global state in a simulation. For that purpose there is one special object that we call *root*. It is the only object that can be explicitly globally referenced. Every simulation has a root object, event-though it might be unused. Default root object is empty, has no properties and no slots.

The idea was rejected because we want to work with local interactions only. Having a global state would open possibility to compromise the local interaction principle.

Counters

Counters were quantitative properties of an object. The quantity stored is a number of instances of countable quality associated with the object. A counter can be imagined as a container able to hold multiple copies of the same tag. The only difference is that the *counter* is a static property of an object.

Counters, as originally designe, were static properties can not be dis-associated from neither associated with an object during run time. Counters can be changed by incrementing or decrementing their values. Counters can be cleared to be zero and they can be tested whether they are zero.

Counters were temporarily rejected because they can be partially implemented through existing mechanisms. Whether counters should remain in the model or not is still open for dicsussion and more research is needed.

Appendix: Symbols used

Table 3: Symbols.

Symbol	
A	actuator
A^1	unary actuator
A^2	binary actuator
G	graph
Γ	selector
M	model
m	subject mode
μ	transition target sepcifier

Symbol	
μ^1	unary transition target specifier
μ^2	binary transition target specifier
n	signal
o	object
O	set of objects
p	symbol presence
Π	selector pattern
R	set of relationships (graph edges)
S	set of symbols
s	symbol - any type
S^t	set of symbols of type t
s^t	symbol of type t
σ	selector
σ_l	selector for left subject
σ_r	selector for right subject
t	symbol type
T	transition
T^1	unary transition
T^2	binary transition
T^2_l	binary transition for left target
T^2_r	binary transition for right target