PRINGL – A Domain-Specific Language for Incentives in Social Computing
§
Supplement Materials

Ognjen Scekic, Hong-Linh Truong, and Schahram Dustdar

Distributed Systems Group, Vienna University of Technology
{oscekic,truong,dustdar}@dsg.tuwien.ac.at
http://dsg.tuwien.ac.at

1 Built-in Operators

PRINGL provides a number of operators for manipulating time, workers and their relationships:

- *Set operators.* – Union, intersection and complement on `Collection<T>`.
- *Time operators.* If working with adjustable intervals, it is advisable to use operators wherever possible as they are evaluated at run time and guarantee that any external changes (e.g., deadline extensions) will be taken into account. A common use-case would see a user initializing an `Interval` from an iteration, and using interval operators to specify points in time in which an action is needed. Time operators are commonly used with temporal specifiers.
  - `StartOf(Interval i)` – returning the `Collection<PoiT>` containing a single time point representing the interval's currently expected starting time.
  - `EndOf(Interval i)` – returning the `Collection<PoiT>` containing a single time point representing the interval's currently expected ending time.
  - `PartOf(Interval I, double p)` – $p[0,1]$ returning the PoiT at percentage p of the interval. $PartOf(i,0) == StartOf(i) PartOf(i,1) == EndOf(i)$
  - `MultiPoint(Interval i, int k)` – returns a `Collection<PoiT>` of points evenly distributed between StartOf() and EndOf().
  - `AllOf(Interval i)` – returns a `Collection<PoiT>` of points representing all time points (depending on the resolution of the underlying system) contained in the interval.
- *Temporal specifiers.* These are special operators used to instruct the execution environment when to perform certain actions or evaluate predicates. As such, they cannot be directly used in user-provided programming code, but are rather offered as a choice through a visual GUI element (drop-down

box) where needed. Internally, they are represented as built-in functions that operate on a collection of `PoiT`s that is provided by the environment at runtime.

- `Always(Collection<PoiT>)` – "at each `PoiT` in collection".
- `Sometimes(Collection<PoiT>)` – "at least once in collection".
- `Once(Collection<PoiT>)` – "exactly once in collection".
- `Never(Collection<PoiT>)` – "never in collection".
- `First(Collection<PoiT>)` – "oldest in collection".
- `Last(Collection<PoiT>)` – "newest in collection".

– *Structural operators.* They perform structural queries/modifications by examining/re-chaining the relationships between worker nodes in the abstraction interlayer (graph) model by using *graph transformations*[1] [1].

- Querying:
  - `neighborsOf(Worker w, string relationType, int numHops, bool directed)` – returns a `Collection<Worker>` filled with workers `numHops` hops away from `Worker` w over un-/directed `relationType` relationships.
  - `managersOf(Worker w)` – returns `Collection<Worker>` filled with manager(s) of worker W. The relationship type representing the managerial relation is obtained from the abstration interlayer.
  - `subordinatesOf(Worker w)` – analogous to `managersOf`.
- Modifying:
  - `changeManager(Worker w, string teamLabel)` – rechains the implicitly determined managerial relations within the members of the `tealLabel` team to point to the new manager.

– *Aggregation operators.* They perform calculations on performance metrics or events over a `Collection<PoiT>`s, in a fashion similar to SQL's aggregate functions. The collection of time points over which the operators calculate is provided by the runtime environment at each invocation. They can only be used in predicate logic blocks ⟨P⟩ that are directly or indirectly reachable through declaration relationships originating from a `WorkerFilter` 𝔽⟩ element.

- `@AVG(double` m`)` – returns the average value of the metric m over the given time point collection.
- `@COUNT(string` *evt*`)` – returns the number of occurrences of event *evt* in the timespan delimited by the first and last `PoiT` in the given input collection.
- `@MAX(double` m`)` – returns the largest value of the metric m over the given time point collection.
- `@MIN(double` m`)` – returns the smallest value of the metric m over the given time point collection.
- `@SUM(double` m`)` – returns the sum of the values of the metric m over the given time point collection.

[1] Please note that the list of structural operators is non-exhaustive at the moment and serves purely for demonstrational purposes.

2 Complex Incentive Elements

2.1 Incentive Logic Subtypes

Subtype	Symbol	Environment-provided input	Allowed output	Intended usage
`TimeLogic`	⟨T⟩	all named Intervals, all Workers, reference to global state	`Collection <PoiT>`	To return time intervals/-points at which a predicate should be evaluated or an action performed.
`StructureLogic`	⟨S⟩	reference to the structural model, reference to global state	`Collection <Worker>` for queries: found workers; for transformations: affected ones	To perform graph queries/transformations on the model representing workforce structure and relationships. A transformation ⟨S⟩ is only allowed to be invoked from ⟨A⟩. A query ⟨S⟩ can only be invoked from ⟨P⟩ and ⟨F⟩.
`PredicateLogic`	⟨P⟩	currently evaluated Worker, all Workers, currently evaluated `PoiT`, reference to global state	`bool`	To evaluate whether a predicate holds at given moment.
`FilterLogic`	⟨F⟩	currently evaluated Interval, all named Intervals, currently evaluated Worker, all Workers, reference to global state	arbitrary	To provide business logic for evaluating past worker performance.
`ActionLogic`	⟨A⟩	Workers to be rewarded/punished, reference to global state	`Collection <Worker>` (affected)	To perform rewarding actions over workers or global variables.

Table 1. `IncentiveLogic` subtypes

2.2 Worker Filter Fields

Field	Description
`time_restr`	An optional ⟨T⟩ returning a collection of time points which should be considered when evaluating workers. If omitted, the default value is a collection containing only a single `PoiT` representing the present moment.
`temp_spec`	An optional temporal specifier (Section **??**) determining how to interpret the filter predicate values across different time points. If unspecified, the predicate is evaluated only for the last (most recent) `PoiT` in the collection.
`auxiliary`	An optional ⟨F⟩ that is used to fetch some global metrics needed for worker evaluation, and possibly provide some intermediate results to be used for evaluating the filter predicate.
`predicate`	A required ⟨P⟩ providing the predicate that will be evaluated against each worker in specified time points.

Table 2. `SimpleWorkerFilter` fields.

2.3 Rewarding Action Fields

Field	Description
`filter`	An optional ◁F▷ determining the workers to which to apply the action. If omitted, the worker collection is by default provided by the runtime environment from the output of the original evaluation filters.
`exec_cond`	An optional ◁P▷ establishing whether the currently evaluated worker earned the reward/punishment or not. If omitted, considered 'true' by default.
`exec_times`	An optional ◁T▷ returning `Collection<PoiT>` determining the possible execution points. If omitted, the environment assumes current `PoiT` and executes immediately.
`temp_spec`	An optional temporal specifier further restricting the original collection of execution `PoiT`s. Defaults to `Always()` if omitted.
`delay`	A hidden parameter set by the environment and used for recalculating execution times in composite rewarding actions. It contains a non-negative integer time offset added to the execution `PoiT`s. The actual time unit is determined as the basic time unit of the underlying layer (an RMod *tick* in our case). The default value is zero.
`action_logic`	A mandatory reference to an ◁A▷ element containing the system-specific business logic that invokes the rewarding action.

Table 3. `SimpleRewardingAction` fields.

2.4 Composite Rewarding Action

A `CompositeRewardingAction` definition consists of graphical elements representing instances of previously defined `RewardingAction`s. It must contain exactly one *initial action* a_0, and exactly k_0 *final actions*, where k_0 is the number of a_0's outgoing edges. The elements are connected with directed edges denoting at the same time: a) `Worker` **flow**; and b) **time delay**. There must be no cycles in the graph, i.e., the flow must be a tree with the root in the initial action, with each final action being the leaf. As any other PRINGL composite type, a composite action can also expose propagated or user-defined parameters.

Worker flow. A `RewardingAction` returns *affected* workers and passes them over outgoing edges. Affected workers are those workers on which the action was successfully applied by the underlying system. The definition of a successful application is system-specific. Therefore PRINGL expects the underlying system to acknowledge via abstraction interlayer that the suggested action was accepted and successfully applied to a worker. The passing of workers is similar to that of composite filters. The two major differences are:

1. The absence of graph cycles prevents the union ($\cup$) operation on passed worker sets.
2. Any `RewardingAction` element can decide whether to use the provided input workers, or completely ignore them, and identify the input workers by itself. For example, a `SimpleRewardingAction` does it by initiating the optional `filter` field. This limitation allows the worker flow to be changed at arbitrary places in the composition.

Figure 1 shows an example of `CompositeRewardingAction` definition. It also shows an example of worker-passing. The initial action /A\ is given the

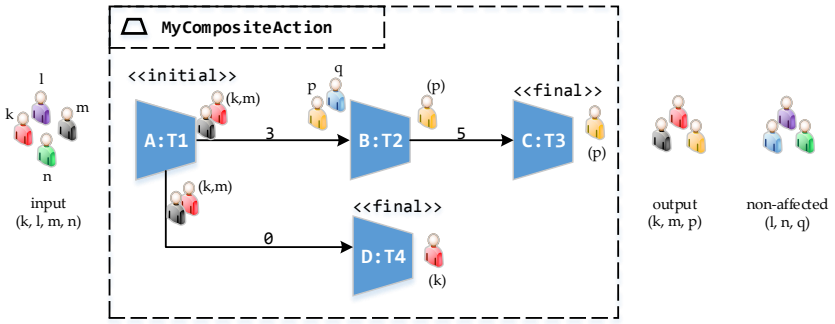

Fig. 1. An example `CompositeRewardingAction` definition.

set (k, l, m, n) as input. The execution of /A\ ends with successful rewarding of workers (k, m). This intermediate set is immediately added to the resulting output set. The same intermediate set of workers is passed to actions /B\ and /D\. Action /D\ ends with rewarding only one of those workers – (k). k is already part of the output, so nothing else happens on this execution branch. The action /B\, on the other hand, discards the input worker set (k, m), and determines its own input set (p, q). After execution, /B\ returns just (p), which is also added to the aggregate output set and passed further as input to /C\, which also happens to award p successfully.

Time delay. Each edge can optionally specify a time delay as a non-negative integer without the unit. If omitted, zero is assumed. The actual unit is determined transparently to the user as the basic time unit of the abstraction interlayer. PRINGL forwards the delay value to the action that the edge point to.

If this action is a `SimpleRewardingAction`, this equals to adding the specified time offset to the hidden `delay` parameter. Later, when executing the action, PRINGL will add the value of the `delay` parameter to each `PoiT` returned by action's `exec_times` ⟨T⟩. If the delay is forwarded to a `CompositeRewardingAction`, then the delay is forwarded to its initial action.

The execution of a composite action starts by first breaking it into linear execution paths containing constituent simple actions. For each execution path PRINGL then takes into account specified delays for each simple action and immediately schedules it with the abstraction interlayer. However, as in this case we need to pass worker sets between actions happening at different times PRINGL needs to store the intermediate results (worker sets) that actions scheduled for a future moment will collect when executed (memoization). In case more than one action is scheduled for execution at the same time, the order is unspecified.

Example. The notion of affected workers is important for incentivizing, because a choice on whether or not to perform a subsequent rewarding action may depend on whether previous actions were successfully applied. Consider a company that wants to reward workers either with free days or with a monetary

reward. The choice is left to the worker. Free days are offered first. Only workers that refuse the free days will be awarded monetary rewards.

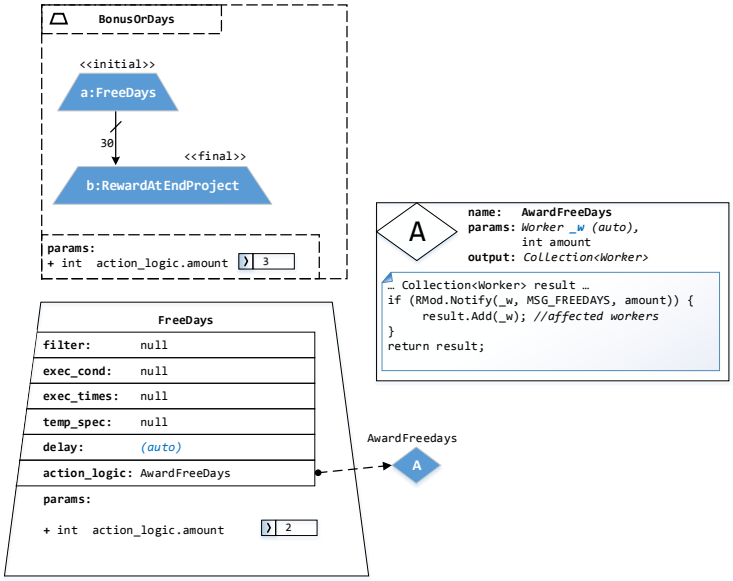

Fig. 2. A `CompositeRewardingAction` letting the workers choose one of the rewards.

We define a new composite rewarding action `BonusOrDays` (Figure 2) that, for the sake of demonstration, assumes the existence of a `RewardAtEndProject` action (similar to the one from the original paper) to award monetary bonuses, as well as a newly-defined action `FreeDays` to award free working days to the workers.

The output of `a:FreeDays` is the set of workers who accepted the 3 free days offered. However, due to a complement edge ($\nrightarrow$) connecting `a` and `b`, the output set of `a` is subtracted from the original input set. Therefore, the input of `b:RewardAtEndProject` are only those workers who declined to accept working days as award, and want to be evaluated at the end of project and paid a bonus according to their performance.

Acknowledgments

This work is supported by the EU FP7 SmartSociety project under grant №600854.

References

1. Baresi, L., Heckel, R.: Tutorial introduction to graph transformation: A software engineering perspective. In: Proceedings of the First International Conference on Graph Transformation (ICGT '02). Volume 70., London, UK, Springer (2002) 402–429