Towards a Programming Paradigm for Artificial Intelligence Applications Based On Simulation*

Jörg Pührer

Institute of Computer Science, Leipzig University
Leipzig, Germany
puehrer@informatik.uni-leipzig.de

Abstract

In this work, we propose to use simulation as a key principle for programming AI applications. The approach aims at integrating techniques from different areas of AI and is based on the idea that simulated entities may freely exchange data and behavioural patterns. We introduce basic notions of a simulation-based programming paradigm and show how it can be used for implementing different scenarios.

1998 ACM Subject Classification I.2 Artificial Intelligence

Keywords and phrases Artificial Intelligence, Simulation, Programming Paradigm

Digital Object Identifier 10.4230/OASIcs.ICCSW.2014.66

1 Introduction

We sketch a programming paradigm based on simulation that is targeted towards applications of artificial intelligence (AI) [13]. Simulation has been used in different fields of AI (such as agent-based systems [10, 14] or evolutionary computation [3]) for achieving intelligent behaviour. The rationale is that many aspects of intelligent behaviour are complex and not well understood but emerge when the environment in which they occur is simulated adequately. We propose to use a simulation environment for realising AI applications that offers an easy way to integrate existing formalisms including methods from different areas of AI such as computational intelligence, symbolic AI, or statistical methods. This environment is composed of interacting *entities* that are grouped in different *worlds* and driven by concurrent *processes*. The proposed approach exploits advances in hardware and concurrent computing due to which simulation became feasible for many applications.

The next section introduces the basic notions of a simulation-based programming paradigm and Section 3 discusses how to use it for modelling different scenarios. In Section 4, we give considerations on the user interface, consistency maintenance, and discuss the relation to related techniques. Section 5 concludes the paper and gives an outlook on future work.

2 Simulation-Based Programming

In this section we explain the proposed simulation-based programming paradigm (SBP) on an abstract level. An SBP system deals with different *worlds*, each of which can be seen as a different point of view. The meaning of these worlds is not pre-defined by SBP, e.g., the programmer can decide to take an objectivistic setting and consider one world the designated

* A longer version of this paper appeared in the proceedings of ReactKnow 2014 [12]. The German Research Foundation (DFG) partially supported this work under grants BR-1817/7-1 and FOR 1513.

real one or treat all worlds alike. Different worlds allow, e.g., to model the beliefs of different agents as in an agent-based approach. Other applications are hypothetical reasoning or realising different granularities of abstraction for efficiency (see Section 3).

A world contains a set of named *entities* which are the primary artifacts of SBP and represent the subjects of the simulation. Entities may have two sorts of named attributes: *data entries* which correspond to arbitrary data (including references to other entities) and *transition descriptions* which define the behaviour of the entities over time. The name of an entity has to be unique with respect to a world and serves as a means to reference the entity, however the same entity may appear in different worlds with potentially different attributes and attribute values. Transition descriptions can be seen as the main source code elements in the approach and they are, similar to the data entries, subject to change during runtime. This allows for a dynamic setting in which the behaviour of entities can change over time, e.g., new behaviour can be learned, acquired from other entities, or shaped by evolutionary processes. We do not propose a particular language or programming paradigm for specifying transition descriptions. It might, on the contrary, be beneficial to allow for different languages for different transition descriptions within the same simulation. E.g., a transition description implementing sorting can be realised by some efficient standard algorithm, while another transition description that deals with a combinatorial problem with many side constraints uses a declarative knowledge representation approach like answer-set programming (ASP) [8, 11] in which the problem can be easily modelled. We require transition descriptions—in whatever language they are written—to comply to a specific interface that allows us to execute them in asynchronous *processes*. In particular, the output of a transition contains a set of updates to be performed on worlds, entities, data entries, and transition descriptions. When a transition has finished, per entity, these changes are applied in an atomic transaction that should leave the entity in a consistent state (provided that the transition description is well designed). As mentioned, transition descriptions are executed in processes. Each process is associated with an entity and runs a transition description of this entity in a loop. A process can however decide to terminate itself or other processes at any time, initiate other processes, and wait for their results before finishing their own iteration.

We now formally summarise the concepts discussed above. We assume the availability of a set Σ of *semantics* for transitions descriptions. A semantics is a function defining how a piece of source code is interpreted and stands for the formalism used for a particular transition description. Let $\mathcal{N}$ be a set of names. We call a concept c *named* if it has an associated name $n_c \in \mathcal{N}$. A *map* is a set M of pairs $\langle n_v, v\rangle$ where v is a named object and $\langle n, v_1\rangle, \langle n, v_2\rangle \in M$ implies $v_1 = v_2$. With slight abuse of notation we write $v \in M$ for $\langle n_v, v\rangle \in M$.

▶ **Definition 1.**

- A *transition description* is a pair $t = \langle \mathrm{sc}, \sigma\rangle$, where sc is a piece of source code, and $\sigma \in \Sigma$ is a semantics.
- A *process* is a tuple $p = \langle t, \mathtt{t}_b\rangle$, where t is a transition description and $\mathtt{t}_b$ is a timestamp marking the begin of the current transition.
- An *entity* is a tuple $e = \langle D, T, P\rangle$, where D is a map of named data, T is a map of named transition descriptions, and P is a map of named processes. Entries of D,T, and P are called *properties* of e.
- A *world* is a map of named entities.
- An *SBP configuration* is a map of named worlds.

For space reasons we describe the runtime behaviour of an SBP system only on an informal level and refer the interested reader to a longer version of this paper [12]. We assume a pre-specified set Υ of *updates* describing what changes should be made to an SBP configuration together with a fixed *update function* υ that maps an SBP configuration, an entity name, the name of a world, and a set of updates, to a new SBP configuration. Whenever a process has finished, it returns a pair $\langle U, b_c\rangle$ where $U \subseteq \Upsilon$ is a set of updates and b_c is one of the boolean values *true* or *false* that decides whether the process should continue with another transition. Intuitively, $\langle U, b_c\rangle$ is the result of the semantics of the transition description of the process given the current SBP configuration. All updates created at one time instant are collected and used as input for the update function υ that produces a follow-up SBP configuration. Consequently, an SBP system can be started with an initial SBP configuration c^0 whose active processes will trigger a sequence of SBP configurations $c^1, c^2, \ldots$ Note that we do not make assumptions whether time is continuous or discrete.

Using names (for worlds, entities, and properties) allows us to speak about concepts that change over time. E.g., if e_0 is an entity $e_0 = \langle D, T, P\rangle$ in some world at time 0 and some data is added to D for time 1 then, technically, this results in another entity $e_1 = \langle D', T, P\rangle$. As our intention is to consider e_1 an updated version of e_0 we use the same names for both, i.e., $n_{e_0} = n_{e_1}$. In the subsequent work we will sometimes refer to concepts by their names and use a path-like notation using the .-operator for referring to SBP concepts, e.g., we refer to an the entity e in a world w by $n_w.n_e$.

3 Modelling AI Applications in SBP

Next, we sketch how to model different scenarios in SBP. Note that we use high-level pseudo code for expressing transition descriptions and emphasise that in an implementation we suggest to use different high-level programming languages tailored to the specific needs of the task handled by the transition description.

In contrast to objects as in object-oriented programming (OOP) entities are not grouped in a hierarchy of classes. Classes are a valuable tool when clear structures and rigorously defined behaviour are required. However, they also impose a rigid corset on their instances: data and behaviour of objects is limited to what is pre-defined in their class. In the scenarios we target, the nature of entities may change and the focus is on emerging rather than predictable behaviour. E.g., a town may become a city and a caterpillar a butterfly, etc., or, in fictional settings (think of a computer game) a stone could turn into a creature or vice versa. We want to directly support metamorphoses of this kind, letting entities transform completely over time regarding their data as well as their behaviour. Instead of using predefined classes, type membership is expressed by means of properties in SBP, e.g., each entity n_e may have a data entry $n_e.types$ that contains a list of types that n_e currently belongs to.

▶ **Example 2.** We deal with a scenario of a two-dimensional area, represented by a single SBP world w, where each entity may have a property *loc* with values of form $\langle X, Y\rangle$ determining the coordinates of the location of the entity. The area is full of chickens running around, each of which is represented by an entity. In the beginning, every chicken *ch* has a transition description *ch.mvRand* that allows the chicken to move around with the pseudo code:

```
wait(randomValue(1..5000))
switch{randomValue(1..4)}
 case 1: return {'mv_up'}; case 2: return {'mv_right'};
 case 3: return {'mv_down'}; case 4: return {'mv_left'}
```

The transition first waits for a random amount of time, and randomly chooses a direction for the move, represented by the updates $mv\_up, mv\_right, \cdots \subseteq \Upsilon$. The semantics of *mvRand* always returns $\langle U, true \rangle$, where U contains the update (the direction to move) and *true* indicates that after the end of the iteration there should be another. When the update function υ is called with one of the *mv* updates it changes the value of *ch.loc*, e.g., if *ch.loc* has value $\langle 3, 5 \rangle$ at time $\mathtt{t}$ and the update is *mv_left* then *ch.loc* has value $\langle 2, 5 \rangle$ at $\mathtt{t} + 1$.

Besides chickens, the area is sparsely strewn with corn. Corn does not move but it is eaten by chickens. Hence, each chicken *ch* has another transition description *eat*:

```
if there is some entity en in myworld with en.loc = my.loc and
  en.types contains 'corn' then return {'eatCorn(en)'}
```

Using the keyword `my` we can refer to properties of the entity to which the transition description belongs (*ch* in this case). Furthermore, `myworld` refers to the world in which this entity appears. Also here, every iteration of *eat* starts another one. For an update *eatCorn*(*en*), the location entry *en.loc* of the corn is deleted and the corn entity is notified about being eaten by setting the data entry *en.eatenBy* to *ch* (we need the information which chicken ate the corn later) and adding a process to the corn entity with the transition description *en.beenEaten*. For now we assume that it only causes the corn to delete itself.

For an initial SBP configuration $c^0 = \langle w \rangle$, where every chicken entity in w has an active process named *move* for transition description *mvRand* and a process for *eat*, an SBP run simulates chickens that walk around randomly and eat corn on their way.

While Example 2 illustrates how data is changed over time and new processes can be started by means of updates, the next example enriches the scenario with functionality for learning new behaviour.

▶ **Example 3.** We extend the scenario of Example 2 to a fairy tale setting by assuming that among all the corn entities, there is one dedicated corn named *cornOfWisdom* that can make chickens smarter if they eat it. It has a transition description *cornOfWisdom.moveSmart*:

```
wait(randomValue(1..1000))
en is an entity in myworld where en.types contains 'corn' and  there is no
 other entity en' in myworld where en'.types contains 'corn'  and
 distance(en'.loc,my.loc) < distance(en.loc,my.loc)
let my.loc=(myX,myY) and en.loc=(otherX,otherY)
if |otherX - myX| > |otherY - myY| then
  if otherX - myX > 0 then return {'mv_right'} else return {'mv_left'}
 else
  if otherY - myY > 0 then return {'mv_down'} else return {'mv_up'}
```

Intuitively, this transition causes an entity to move towards the closest corn rather than walking randomly as in *mvRand*. Another difference is that *moveSmart* processes have shorter iterations on average as the range of the random duration to wait is smaller. The *cornOfWisdom* does not have active processes for this transition definition itself but can pass it on to everyone who eats it. This is defined in the transition *cornOfWisdom.beenEaten* that differs from the *beenEaten* transition description of other corn by not only deleting the corn but also copying the *moveSmart* transition description from the *cornOfWisdom* to the chicken by which it was eaten. It also changes the *move* process of the chicken to use its new *moveSmart* transition description instead of *mvRand*. Thus, if a chicken happens to eat the *cornOfWisdom* it subsequently has a better than random strategy to catch some corn.

Having means to replace individual behavioural patterns, as in the example allows for modelling evolutionary processes in an easy way. E.g., if the chicken scenario is modified in a way that chicken which do not eat corn regularly will die, a chicken that ate the *cornOfWisdom* has good chances to survive for a long period of time. Further processes could allow chickens to reproduce when they meet such that baby chicken may inherit which transition description to use for moving from one of the parents. Then, most likely, chicken using *moveSmart* will be predominant soon.

Example 3 also illustrated that inheritance in SBP works on the individual level: entities can pass their transition descriptions and data entries to fellow entities. Thus, compared to OOP, inheritance is not organised in a hierarchical way. The underlying motivation is to follow the main idea of simulating real world objects, taking the stance that entities in nature are individuals that are not structured into distinct classes per se.

The next example illustrates the use of worlds for hypothetical reasoning.

▶ **Example 4.** Entity *barker* represents a waiter of a restaurant in a tourist area trying to talk people on the street into having dinner in his restaurant. To this end, *barker* guesses what food they could like and makes offers accordingly. We assume an SBP configuration in which for every entity h that represents a human there is a world w_h that models the view of the world of this person. The transition description *barker.watchPeople* allows *barker* to set the eating habits of passer-by in his world w_{barker} using country stereotypes:

```
wait(randomValue(50))
let en be an entity in myworld where en.loc near my.loc,
  en.types contains 'human', and en.eatingHabits = unknown
 country = guess most likely home country of en
 prototype = myworld.country.inhPrototype
 return {'setEatingHabits(en,propotype)','setPotentialCustomer(en)'}
```

For every country w_{barker} contains a reference *inhPrototype* to an entity representing a typical person from this country. The update *setEatingHabits*$(p1, p2)$ copies transition descriptions and data properties related to food from person $p2$ to person $p1$. The update *setPotentialCustomer*(p) lets *barker* consider entity p to be a potential customer. In order to choose what to offer a potential customer, the waiter thinks about what kind of food the person would choose (based on his stereotypes). This is modelled by transition *barker.makeOffer*:

```
let cus be a potential customer in myworld
w' = copy of myworld
w'.cus.availableFood = restaurant.availableFood
w'.cus.hungry = true
intermediate return {addWorld(w'), startProcess(w'.cus.startDinner)}
when process w'.cus.foodSelected is finished
 food = w'.cus.selectedFood
 return {praiseFood(food), deleteWorld(w')}
```

To allow for hypothetical reasoning by the waiter, a temporary copy w' of world w_{barker} is created. The sole purpose of w' is to simulate the customer dining. We need a temporary world as we use the same transition descriptions that drive the overall simulation. E.g., if we would use w_{barker} instead, it would mean that *barker* thinks that the customer is actually dining, using the world of the customer would mean that she thinks she is dining and so on.

After creating w', the transition description defines that the food available to the version of the customer in w' is exactly the food that is on the menu of the restaurant and the customer is set to be hungry in the imagination of the waiter. Then, w' is added to the SBP configuration and a process for $w'.cus$ is started using the transition description *startDinner* that lets enitity *cus* start dining in w'. Note that the keyword `intermediate return` in the code is a convenience notation that allows for manipulating the SBP configuration during a transition which is strictly speaking not allowed in our framework. The same behaviour could be accomplished in a complient way by splitting *makeOffer* into two transition descriptions used in an alternating scheme. As soon as the customer chooses some food in the simulation, transition *barker.makeOffer* is notified which then reads which food has been chosen in the hypothetical setting. Finally, the update *praiseFood*(*food*) causes the waiter to make an offer for the chosen food, whereas *deleteWorld*(w') deletes the temporary world.

Note that copying worlds as done in Example 4 does not necessarily imply copying all of the resources in this world within an implementation of an SBP runtime engine (cf. Section 5).

▶ **Example 5.** We continue Example 4 by assuming an SBP configuration where a tourist, *Ada*, passes by the waiter. His process for transition *barker.watchPeople* classifies Ada by her looks to be an Englishwoman. After that, the process for *barker.makeOffer* starts hypothetical reasoning about Ada having dinner. Following the stereotypes of *barker* about English eating habits, the process reveals that *Ada* would go for blood pudding which he offers her subsequently. However, Ada is not interested in this dish as she is vegetarian. She explains her eating habits to the waiter, modelled by the transition description *ada.explainEatingHabits*:

```
let pers be current discussion partner in myworld
if pers offers food containing meat then
 let w_pers be the world of pers
 return {'setEatingHabits(w_pers.me, myworld.me)'}
```

Here, update *setEatingHabits* that we know from Example 4 overwrites food related properties of the entity representing Ada in the world of *barker* with her actual eating habits. If *barker* repeats the dining simulation for making another offer the result matches her real choices.

The last two examples showed how worlds can be used to express different modalities like individual points of views or hypothetical scenarios. Next, we sketch a setting where different worlds represent the same situation at different granularities.

▶ **Example 6.** Consider a computer game in which the player controls a character in an environment over which different villages are distributed. Whenever the character is close to or in a village the inhabitants of the village should be simulated following their daily routines and interacting with the player. However, as the game environment is huge, simulating all inhabitants of each village at all times is too costly. The problem can be addressed by an SBP configuration with two worlds, $w(v)_{act}$ and $w(v)_{apx}$, for each village v. Intuitively, $w(v)_{act}$ simulates the village and its people in detail but has only active processes while the player is closeby. The world $w(v)_{apx}$ approximates the behaviour of the whole village, e.g., increasing or shrinking of the population, economic output and input, or relations to neighbour villages, based on statistics and it has only active processes when the player is not around. When the player enters a village, a process is started that synchronises the world $w(v)_{act}$ with the current state of the village in $w(v)_{apx}$, e.g., by deleting or adding new inhabitants or shops. It also starts processes in $w(v)_{act}$ and cancels processes in $w(v)_{apx}$. If the player leaves again, another type of process performs an opposite switch from $w(v)_{apx}$ to $w(v)_{apx}$ being active.

4 Discussion And Related Work

For handling interaction of an SBP system with the user or another system, we suggest to use externally controlled processes: A transition description can use a dedicated 'external semantics' where the result returned for every transition is provided externally. By deciding which parts are controlled externally on the level of transition descriptions, the behaviour of an entity can be partially controlled by the user and partially by the system. This way one could replace, e.g., a human player in a computer game by an AI player or vice versa.

A natural question is how concurrent access on data is handled in the approach. Conceptionally, conflicting updates that occur at the same time instant are managed by the update function v. In practice, however, this function has to be implemented and conflicts have to be addressed. Here, techniques for concurrency control in databases [1] could be useful. Besides technically conflicting updates on data, another issue is semantical inconsistency, i.e., data whose meaning with respect to the modelled problem domain is conflicting. Consider an SBP configuration modelling a banana and two monkeys where each monkey has a transition description for grabbing the banana. Suppose that both monkeys detect the banana their grabbing processes start. Then, it could happen that the system gets into a state where each monkey is believed to have the banana. It is probably sensible to tackle consistency problems of this kind on the level of modelling rather than by the underlying computational framework as there are many types these issues that must be addressed differently and also in the real world two monkeys could believe that they succeeded in getting a banana for a moment.

One goal of SBP is integrating different AI techniques where a main mechanism of integration is the explicit use of existing AI formalisms (e.g., ASP, planning techniques, etc.) for solving emergent subproblems by means of transitions descriptions. This is related to the use of different contexts in recent reactive forms of heterogenous multi-context systems [2, 4].

Evolutionary processes are intrinsic to many types of simulation. In, genetic algorithms [5, 9] the fitness of individuals is typically rated by a dedicated fitness function, whereas the most obvious approach in SBP is simulating natural selection by competition in the simulated environment, as discussed for the chicken scenario after Example 3. Evolving behaviour is related to genetic programming [6] where programs are shaped by evolutionary processes.

Agent-based systems (ABS) [10, 14] share several aspects with SBP like the significance of emerging behaviour when entities are viewed as agents. This view, however, is not adequate for all types of entities, e.g., when they represent objects like stones, collections of entities, or intangible concepts like 'the right to vote' which should not be seen as agents. Moreover, agents interact via explicit acts of communication that can but need not be modelled in an SBP configuration. Thus, we see SBP conceptionally one level below ABS, i.e., SBP languages can be used for implementing ABSs rather than being ABSs themselves. Nonetheless, many techniques from ABS literature could be beneficial for future SBP systems.

The idea of using multiple worlds for different points of view and modalities is loosely related to the possible world semantics of modal logics [7].

5 Conclusion

We proposed initial ideas on an approach for using simulation as a programming paradigm. Its cornerstones are 1. typeless entities, 2. different worlds for different views on reality, 3. behaviour defined by heterogenous concurrent services, and 4. exchange of behavioural patterns and individual inheritance. Clearly, there are many important aspects that were not discussed here that need to be addressed when putting SBP in practice. Thus, a major goal is a prototype of an SBP engine which opens a wide range for further work: E.g., on how to manage resources in SBP systems in which many worlds and entities share slightly different data and processes. Efficiency requirements could necessitate mechanisms for sharing resources, e.g., by only keeping track of differences when a world or entity is cloned.

References

1 Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

2 Gerhard Brewka, Stefan Ellmauthaler, and Jörg Pührer. Multi-context Systems for Reactive Reasoning in Dynamic Environments. In *Proceedings of the 21st European Conference on Artificial Intelligence* (*ECAI 2014*), pages 159–164, 2014.

3 Lawrence J. Fogel, Alvin J. Owens, and Michael J. Walsh. *Artificial Intelligence through Simulated Evolution*. Wiley, Chichester, WS, UK, 1966.

4 Ricardo Gonçalves, Matthias Knorr, and João Leite. Evolving Multi-context Systems. In *Proceedings of the 21st European Conference on Artificial Intelligence* (*ECAI 2014*), pages 375–380, 2014.

5 John H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, 1975.

6 John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

7 Saul Kripke. A Completeness Theorem in Modal Logic. *Symbolic Logic*, 24(1):1–14, 1959.

8 Victor W. Marek and Mirosław Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In Krzysztof R. Apt, Victor W. Marek, Mirosław Truszczyński, and David S. Warren, editors, *In The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer, 1999.

9 Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

10 Muaz Niazi and Amir Hussain. Agent-based Computing from Multi-agent Systems to Agent-based Models: a visual survey. *Scientometrics*, 89(2):479–499, 2011.

11 Ilkka Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.

12 Jörg Pührer. Towards a Simulation-based Programming Paradigm for AI applications. In *Proceedings of the International Workshop on Reactive Concepts in Knowledge Representation (ReactKnow'14)*, pages 55–61, 2014.

13 Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

14 Michael J. Wooldridge. *An Introduction to MultiAgent Systems (2. ed.)*. Wiley, 2009.