Actions, Objects, and Subjects

Hannu-Matti Järvinen Department of Pervasive Computing, Tampere University of Technology, Finland

Abstract— *This paper proposes an action-oriented computational model to be used as the low-level implementation of programs, hence effectively omitting processes. The benefits of the model are that concurrent execution, mutual exclusion, and synchronisation are automatically provided without explicit programming, messages are not needed between executing bodies, deadlocks do not exist at the action level, and the system uses implicitly as many parallel execution units as possible.*

However, action orientation is not a natural way to think for humans. Therefore, also subjects, which are objects providing liveness, are introduced to make programming sequential, but still making it easy to implement the program as actions, hence getting the benefits listed above.

Keywords: Concurrent languages, concurrent programming, parallel processors

1. Introduction

Concurrent execution has become a more and more important part of software. Many systems are distributed and processors have several cores. Traditionally, each process computes one task, and in some cases, interacts with other processes or threads. This interaction may take place as common variables, causing critical sections to appear, or by message passing, where messages carry data from process to process. Anyway, introducing concurrency causes needs for mutual exclusion and synchronisation mechanisms, and creates problems like starvation and deadlock.

To overcome the problems caused by concurrency, programming languages offer concepts like monitors, message passing, channels, futures, or rendez-vous. However, the basic problems, i.e., mutual exclusion and synchronisation, still exist, and the programmers have to understand them to cope with them. In most cases, concurrency has to be programmed explicitly into programs, the main exception being parallel processing in scientific computing.

The action-oriented execution model used in this paper was independently suggested by Chandy and Misra as Unity [2], by Back and Kurki-Suonio as Joint Actions [1], and by Lamport as Temporal Logic of Actions (TLA) [8]. The model makes critical sections obsolete, and hence also mutual exclusion. The model also has a built-in synchronisation mechanism. These are evident benefits if compared with the traditional process-oriented execution model. However, although this model has been used in several research projects, the models have not been adopted in practical software production. There are probably two main reasons for that. First, humans tend to think causally and find causal relations even if there are none [6]. Using actions is in contradiction to this causality. Second, all these methods are intended for specification and the real implementation still has to be process-oriented. Since changing the program structure from action-oriented to process-oriented is not a trivial task and there are no tools to automatically do it, this may consume all benefits that action orientation can offer. In short, the action-oriented paradigms have been suggested for high-level specification, and the implementation is assumed to be process-oriented.

In this paper, a low-level action-oriented system is introduced with high-level sequential programming bodies, *subjects*. The benefits of the idea are that synchronisation and mutual exclusion is implicitly solved by the action-oriented model, but the programmer will still be able to describe the system as programs where statements have the human-expeced causal relationships.

This paper is organised as follows: First, action system with objects are discussed, then action execution at low level is introduced, and before brief discussion subjects are described through an example.

2. Action-oriented execution model with objects

In action-oriented models, actions can be considered either state transitions from a state to another, or relations between two states. Hence, actions do not contain any memory, but the whole state of the computation is stored in variables. In contrast, in process-oriented systems the state of the computation is in the variables and program counters of the processes or threads. The DisCo specification language uses the same model as [1], [2], and [8], but its variables are collected into objects [4]. The action-oriented approach is discussed in detail by Kurki-Suonio in [7].

2.1 Concurrent Actions

Actions alter the state, but do not consume time ideally. In TLA, an action A can be expressed as a predicate between two states, e.g. $s[[A]]t$, where s and t are states. Starting from a state s_0, we can see a sequence of actions take place, forming a *behaviour*. Formally, a behaviour is an infinite sequence of states $\sigma = s_0, s_1, s_2, s_3...$

There is a non-stuttering[1] action between each state of

[1]Stuttering actions do not alter the interesting set of variables; they can be added anywhere. Altering the set of interesting variables, alters the set of stuttering actions, too, and an alternative behavior is constructed.

a behaviour. If two actions access distinct sets of variables, their order can be changed without violating TLA formulas, since TLA does not have operators *next step* or *previous step*. Hence, behaviours $\sigma_0 = s_0, s_1, s_2, s_3, ...$ and $\sigma_1 = s_0, s_2, s_1, s_3, ...$ are different, but there are no TLA predicates that can distinguish these behaviours from each other (note that if the sets of accessed variables are not distinct, this is probably not the case, but then action guards may prevent either behaviour).

In action-oriented models, therefore, concurrency is modelled by nondeterministic selection of actions to be executed. Further, actions accessing distinct variables can be executed concurrently, since we can serialise them in any order. To be precise, we can execute actions concurrently, if the sets of assigned variables are distinct and action do not refer to variables assigned to by other actions. Formally, if V_{A_a} is the set of variables assigned (and referred) to by action A, and V_{A_r} is the set of variables referred only by action A, then parallel execution is possible for a set of *distinct* actions $\mathbf{S}$, where

$$\begin{aligned}&\forall A, B \in \mathbf{S}, A \neq B : \\ &V_{A_a} \cap V_{B_a} = \emptyset \wedge V_{A_a} \cap V_{B_r} = \emptyset \wedge V_{B_a} \cap V_{A_r} = \emptyset \end{aligned} \quad (1)$$

This means that even if the action models are originally sequential, we can execute actions concurrently, provided that (1) holds for each set of actions executed concurrently. Since actions do consume time in real computation, this might have other consequences as well, but they are not discussed in this paper. When variables are collected into objects, (1) can be used for objects instead of variables. In some cases, this may appear as unnecessarily strict, but this is good enough for many practical purposes.

In this paper, the action-oriented execution model is used as the lowest mechanism in computation instead of processes. Since the model allows concurrent execution of distinct actions, we get a system which can use as many parallel processing units as there are distinct actions enabled. There are huge benefits in this idea: without the concept of processes, programmers do not need to take care of mutual exclusion or synchronisation, deadlocks cannot happen at action level, and there is no need for message passing. All execution takes place in actions that alter the contents of objects that participate in them.

The basic components of the action-oriented execution model are objects and actions. In this paper, we use a model derived from DisCo [5]. Actions, objects, and their properties in this model are briefly introduced.

2.2 Objects

An object may contain any kind of data, data structures, containers, or combination of them. In this paper, objects are unnamed entities that contain data; from the programmer's point of view they can be considered record types.

Objects are usually created by the initialisation of the system. However, the model allows the creation and destruction of objects and actions during execution.

2.3 Actions

An action can be considered a potential state transition between two states determined by the objects of the system. The action has participants (objects) that participate in the action in *roles*, and two parts: a *guard* and a *body*. The guard indicates when the action can be selected for execution, i.e., the action is *enabled*. The body contains a piece of program code that is executed whenever the action is selected for execution. The participants are distinct, i.e., an object may participate in an action in one role only. In contrast to DisCo, syntactic actions are not models that introduce a set of TLA actions whose participants are computed in run-time, but actions have fixed participants and parameters that are given at the creation time.

Action guards and bodies may refer to the data of the action participants only. Any datatype can be part of an object. Since actions may not refer to data outside its participants, referencing through pointers or references is allowed only in those cases where the target data is also either a participant of the action or part of a participant object.

The action guards in Joint actions, Unity, TLA, and DisCo can be divided into three parts:

1) *Global part* that can refer to any variable or object in the system. The reference is possible by quantifications or closures.
2) *Common part* that can refer to the contents of several participants.
3) *Local part* that can refer to the contents of a single participant only.

Although a very powerful expression, global part of the guard is hard to implement and very inefficient if implemented. They are not included in the action model of this paper. Furthermore, action guards are syntactically separated to local and common parts to make the implementation of the scheduler more efficient. For example, the local part of the guard has to be evaluated only, if the contents of the participant are changed, and if any of local parts of the action guard is false, there is no use to evaluate the common part. Once evaluated, the value of the common part can be stored for further use until the contents of an action participant is changed.

2.4 Objects in Action-oriented Systems

In traditional object-oriented systems, classes contain both the data and the methods that access the data. This is illustrated on the left side of Figure 1. Whenever the values of objects are needed or updated, corresponding method is invoked. Method calls are indicated by arrows in Figure 1 to

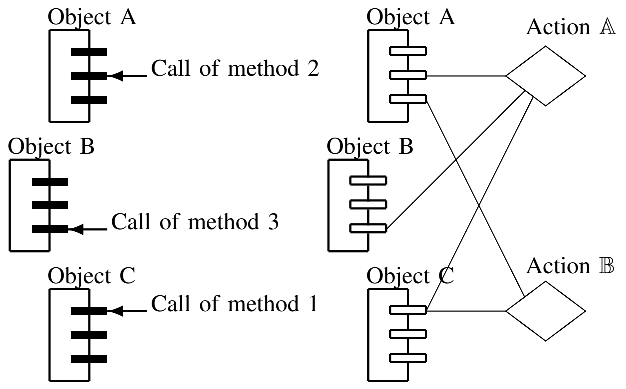

Fig. 1: Classes and methods (left), Classes and actions (right)

emphasise that an external caller is needed. The methods are printed in black, since normally their contents are private.

On the right side of Figure 1, the action $\mathbb{A}$ (illustrated by a large diamond) contains the calls illustrated on the left side of the picture. Action $\mathbb{B}$ is another action with two participants; it shares the first method of object C with action $\mathbb{A}$. Methods are white, since they are programmed as part of the action. Although private implementations of methods could be used also with actions, in any case an action has to contain some code of its own to make the methods interact. An action may be selected for execution if it is *enabled*. Hence, an external caller is not needed and lines are used instead of arrays to emphasise this.

3. Action Execution

This part describes the high-level idea of execution where basic primitives are actions instead of processes. Without going to details, there exists an experimental environment where this model has been tested.

The system consists of a scheduling processor and n processing units that execute the actions. These can be traditional processors. For efficiency, they should be connected to each other by special hardware, but the experimental environment we have is built to run on a normal multiprocessor computer. The overall structure is illustrated in Figure 2.

The model works as follows: First, the scheduler selects a set of enabled actions from the action store to the action queue, remembering that actions to be selected shall be distinct from each other and actions already in the queue or in execution. Second, one of the processing units takes an action description from the queue, loads the action code and accessed objects into its local memory[2], and executes the action. Third, when the execution is finished, altered objects are copied to the common memory, and action is returned to the action store.

[2]Loading is not needed if the processors have access to the common memory.

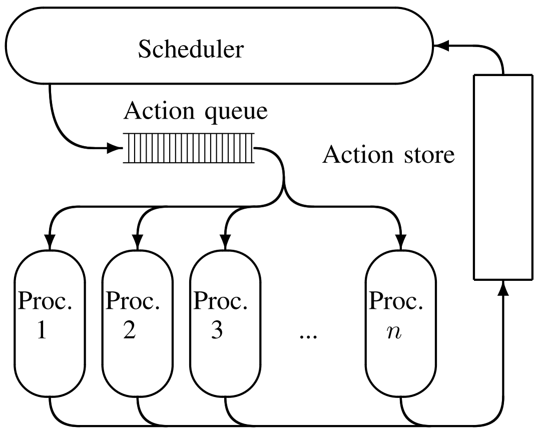

Fig. 2: The overall structure of action execution.

In short, synchronisation conditions are expressed by action guards, and mutual exclusion is automatically taken care of by the scheduler. This means that the basic primitives for concurrent execution are handled by hardware and the scheduler instead of the programmer, much like how the virtual memory is taken care of by hardware and the operating system instead of manual overlay by programmers. Note that the code to be executed is not affected when processors are added or removed.

3.1 Crossing roads

An example of crossing roads is used to show how action-oriented approach works. Consider a crossing of two roads, one north to south and the other west to east.

There is a lane to go forward or turn right and a separate lane of its own to turn left, all of these for each direction. Each lane has a corresponding traffic light, as illustrated in Figure 3.

The lanes are named as follows: the first letter gives the approaching direction, and second and third letters give the destination. So, SW stands for from south to west, and SNE from south to north and east. One possible set of pairs that are safe to have green lights on at the same time is NSW and SNE, WES and EWN, NE and SW, and WN and ES. Running this sequence will eventually give all directions a green light.

There are three classes that can be identified: a traffic light, a lane, and a car. The car is not part of the system but an external object. The lane is an abstraction that is connected to traffic lights: the traffic light can be green or yellow only if the lane is safe, i.e. all lanes crossing the lane have the corresponding traffic light red.

The simplest class is *Lane*. It has to contain a variable indicating if the lane is safe or not. Class *Traf_light* includes

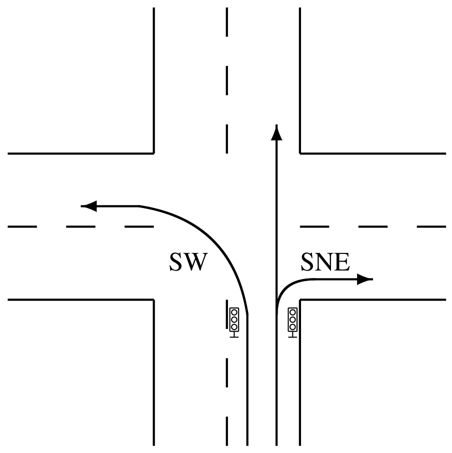

Fig. 3: Crossing with lanes SW and SNE and their signals illustrated.

```
class Lane is
  safe: Boolean:=false;
end;

class Traf_light is
  lamp: (RED, YELLOW, GREEN):=RED;
end;
```

Fig. 4: Classes Lane and Traf_light.

a state variable that corresponds to the colour of the traffic light. Classes *Lane* and *Traf_light* could be combined to one class since they are closely connected, but this would make the objects less intuitive. The classes are given in Figure 4.

Actions *set_red*, *set_green*, *set_yellow*, *free_lane*, and *reserve_lane* are using these classes. The first four actions have participants of the given classes only and they are shown in Figure 5.

Action *set_green* puts the green light on, if the corresponding *Lane* is safe. Action participants are defined between keywords *is* and *body* of the action. Expression after participant name list (after the colon) is the local guard of the participant.

Note that the connection between participants *lane* and *light* could be expressed in the common guard using identities (e.g. lane.id=light.id), but since these relations never change, the condition is left out, and actions are created for only those pairs of lanes and traffic lights that are connected.

Action *set_yellow* is enabled, if the traffic light is green and given time have been elapsed. It has only one participant, *green_on*, indicating the time green light should be on. This value is given when the action is created. Function *timeout* returns true if given time has elapsed since last update of the object. This requires that also the time of the update is stored when updating an object. Action *set_red* is almost

```
action set_green is
  Lane as lane: safe;
  Traf_light as light: lamp=RED;
body
  light.lamp:=GREEN;
end;

action set_yellow(green_on: seconds) is
  Traf_light as light:
    lamp=GREEN and timeout(green_on);
body
  light.lamp:=YELLOW;
end;

action set_red (yellow_on: seconds) is
  Traf_light as light:
    lamp=YELLOW and timeout(yellow_on);
body
  light.lamp:=RED;
end;

action free_lane (margin: seconds) is
  Lane as lane: safe;
  readonly Traf_light as light:
    lamp=RED and timeout(margin);
body
  lane.safe:=false;
end;
```

Fig. 5: Actions set_green, set_yellow, set_red and free_lane.

identical to action *set_yellow*.

Action *free_lane* waits *margin* seconds after the execution of *set_red* to be sure that there are no cars on the lane before freeing it.

To enforce any sequence to go through all lanes, a control object is needed to guide the behaviour of action *reserve_line*. The control class has four states indicating the current phase of the system; this ensures all directions will get green light in a steady basis. The control class, action *reserve_line* are in Figure 6.

3.2 Initialisation

The initialisation code has to create the traffic lights, lanes, and actions. It is intended to be sequential code executed before the action system is invoked. Note that new actions and objects can be created and deleted also in run-time. The code in Figure 7 shows how the participants and parameters of actions are given static values in creation, hence making it possible to omit common parts of guards describing the relations between the objects.

This system will execute forever. In some cases execution will lead to a situation where no actions are enabled and the

```
type Safe_pairs is
  (NSW_SNE, NE_SW, WES_EWN, WN_ES);
class Control is
  state: Safe_pairs:=NSW_SNE;
end;

action reserve_lane
  (current, next: Safe_pairs) is
  Control as cont: state=current;
  readonly Lane as old_1, old_2:
    not safe;
  Lane as new_1, new_2: not safe;
body
  cont.state := next;
  new_1.safe, new_2.safe:=true, true;
end;
```

Fig. 6: Action reserve_line

```
initially
  nsw, sne, ne, sw, wes,
    ewn, wn, es: Lane;  // Creates lanes
  tl_nsw, tl_sne, tl_ne, tl_sw, tl_wes,
    tl_ewn, tl_wn, tl_es: Traf_light;
  control: Control;
  // Some constant values
  green_on, yellow_on: seconds:=20, 5;
  margin: seconds:=8;
  // Create actions for a lane
  create set_red(yellow_on, nse);
  create set_yellow(green_on, nse);
  create set_green(nse, tl_nse);
  create free_lane(margin, nse, tl_nse);
  // etc. for each lane
  create reserve_lane(NSW_SNE, NE_SW,
                      nsw, sne, ne, sw);
  // etc. for each safe pair
end initially;
```

Fig. 7: Initialisation code for the action version

system terminates. This means that deadlocks are possible at the application level although at action level the deadlocks caused by the mutual exclusion of objects as resources are prevented.

4. Subjects

As mentioned in the introduction, the action-oriented view is not close to the way humans think. For example, reading and changing the algorithm that controls which lanes are given turn is not easy. Hence, even if the traditional problems of concurrency and scalability for n processors could be solved by actions, there is no much hope to alter the way of human thinking. Virtual memory was referred to as an analogous example. It also gives a good next goal, since one cannot see from the source code if the application is made for a virtual memory computer or not. Unfortunately, action orientation abandons the basic abstraction of processes and use actions instead and this is too drastic a change to be completely hidden from the programmer.

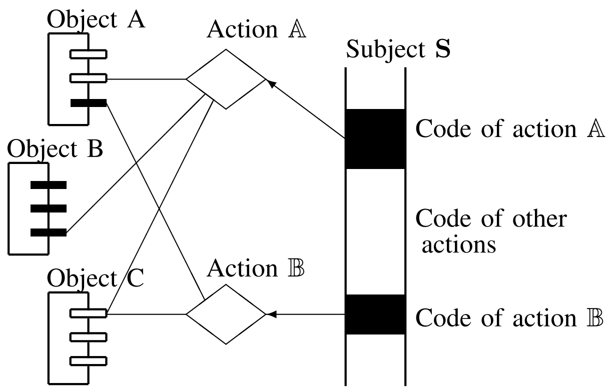

Fig. 8: Subjects versus actions and objects.

To make programming for action systems closer to sequential thinking characteristic for humans, a concept of *subjects* is introduced. Note that although term *subject* is used, the model does not follow subject-oriented programming introduced in [3]. However, there are some similarities. In [3] subjects are different views to an object; in this paper, subjects are different views to a set of objects.

On the left side of Figure 8, there are the same actions and objects as were on the right side of Figure 1. On the right side of Figure 8 is a subject code, from which the actions are recognised and implemented. Note that there can be both private and open methods; however, their details are not it the scope of this paper.

In contrast to action-oriented systems, subjects provide an illusion of sequential behaviour; this is maintained by an implicit control object (not shown in Figure 8) and writing the code sequentially. The compiler recognises the boundaries of actions, and transforming this to an action system can be hidden from the programmer. Concurrency is implicit, if there are more than one subject in the system, and the programmer does not need to worry about the mutual exclusion or synchronisation of objects.

The example of crossing roads is revisited to illustrate how subject-oriented approach works. We can observe this example from three points of view. These views are first described informally as follows.

First, the car (or the driver) sees the crossing as the following sequence of events:

1) Approach the crossing and select correct lane x.
2) Arrive at the stop line of lane x. Stop if light x is red or yellow has been on for some time.

3) Go if light x is green or yellow has just appeared.
4) Leave the crossing.

Since the car is not part of the system, this is an interface of the system, and hence not implemented.

Second, the traffic light for each lane is internal to the system and runs repeatedly the following sequence:

1) Show red light.
2) The lane is safe, show green light.
3) Time out for green, show yellow light.
4) Time out for yellow, show red light.
5) Time out for safety margin, release the lane.

Third, the internal control logic may repeat the following sequence:

1) Wait lanes are free and reserve the first pair of lanes.
2) Wait lanes are free and reserve the second pair of lanes.
3) Wait lanes are free and reserve the third pair of lanes.
4) Wait lanes are free and reserve the fourth pair of lanes.

Naturally, the control logic could be much more sophisticated. For example, each step could check if there are cars approaching on the corresponding lane. However, we continue with this simple system.

We can identify the following objects: cars, traffic lights for each lane, and the lanes. Cars are not part of the implementation, and lanes and lights could be united, but since the lights are actually control objects (subjects) and lanes are mostly abstractions within the system, we keep them separate. Using these objects we can write subjects Traf_light and Control (Figure 9) from the informal description (the cars and initialisation of the system have been omitted to save space). Note that although class *Lane* of Figure 4 is used, the introduction of subject *Traf_light* makes separate class for traffic lights obsolete.

Each wait statement on Control and Traf_light can be represented as an action. For example, the first wait of Control could be transformed to the action *Control_1* in Figure 10 that resembles action *reserve_line* in Figure 6.

The detailed semantics of the programming language used for subjects is not in the scope of this paper. Actually, any object-oriented programming language will do, if *subjects* and statements *wait* and *collection* are added.

The *wait* statement means that the execution of the subject in that branch is stopped until the given condition is true. In translating to actions, the condition is used as the guard of an action and its statements form the action body.

The *collection* statement indicates alternatives that may take place; each of them is actually an action of its own, and if they are distinct, they can be executed in parallel. This statement was not needed in this example.

The following rules to transform sequential code of subjects to actions are created from the notes above.

1) Create a control object with a control variable for the subject.
2) The guard of the first action is a test if the control variable is in the initial state.

```
subject Traf_light (lane: Lane,
  green_on, yellow_on, margin: seconds)
is
  lamp: (RED, YELLOW, GREEN);
begin
  lamp:=RED;
  loop
    wait lane.safe then
      lamp:=GREEN;
    end;
    wait timeout(green_on) then
      lamp:=YELLOW;
    end;
    wait timeout(yellow_on) then
      lamp:=RED;
    end;
    wait timeout(margin) then
      lane.safe:=false;
    end;
  end loop;
end Traf_light;

subject Control (
  p1s, p1t, // pair 1: straight and turn
  p2s, p2t, p3s, p3t, p4s, p4t: Lane) is
begin   // Lanes are initially unsafe
  loop
    wait not (p4s.safe or p4t.safe) then
      p1s.safe, p1t.safe:=true, true;
    end;
    wait not (p1s.safe or p1t.safe) then
      p2s.safe, p2t.safe:=true, true;
    end;
    wait not (p2s.safe or p2t.safe) then
      p3s.safe, p3t.safe:=true, true;
    end;
    wait not (p3s.safe or p3t.safe) then
      p4s.safe, p4t.safe:=true, true;
    end;
  end loop;
end Control;
```

Fig. 9: Subjects Traf_light and Control

```
action Control_1 is
  readonly Lane as p4s, p4t: not safe;
  Lane as p1s, p1t;
  Control as cont: status=1;
begin
  p1s.safe, p1t.safe:=true, true;
  cont.status:=2;
end Control_1;
```

Fig. 10: Action Control_1 generated from subject Control

3) The body of the first action starts immediately after *begin*, and ends when a collection, wait, or a loop containing wait(s) is encountered.
4) A loop containing waits introduces a new control object that is used within the loop, applying these same rules. If the loop does not include waits, it is implemented as a traditional loop in the action body.
5) The condition of a wait becomes the guard of the next action.
6) The code until the next collection, wait, or loop containing wait(s) becomes the body of the action.
7) Add update code for the control variables into each action body, and the corresponding tests of the control variables to the action guards.

Actions can be automatically generated from the subject descriptions and directly executed by the action model computer described earlier as illustrated in Figure 8. The objects participating in actions can be resolved relatively easy by the compiler.

Subjects and objects are closely related. Objects provide room for local variables and methods associated with them; so do subjects, but they also make something happen. When methods describe potential things that can happen, i.e., safety properties, subjects also introduce liveness properties for the system, just like processes have implicit liveness properties.

5. Discussion and Conclusions

The action-oriented model can be used on top of a general purpose operating system as was done when the concept was tested by an experimental environment built on Posix threads interface. The experiments show that the action-oriented system is working in principle, but the existence of the operating system and its scheduler makes it impossible to determine if this paradigm is really competitive in comparison with a traditional process-oriented system. The experiments also indicate that action-oriented programs work logically equally well regardless of the number of processors available.

The model is expected to work better in specialised environments having multiple processors. Hence, actual hardware supporting the action orientation as shown in Figure 2 together with measurements are needed—in other words, the idea needs empirical results to support it.

The main difference with action orientation and most other approaches is to completely omit the idea of processes, and gather data to objects that participate in actions. In spite of the benefits of action systems, they are as such not very attractive, since they are in contradiction to the way humans think. Our thinking is based on causality, and we even tend to find causality when there is none. Hence, working with actions requires a lot of effort. Introducing subjects that describe the system from the viewpoints of active interfaces gives the programmer possibility to think causally, and makes it easy to implement the program as actions.

The proposed approach has several benefits:

1) implicit mutual exclusion
2) implicit synchronisation
3) the prevention of most of the potential deadlocks
4) less starvation possibilities
5) still sequential programming
6) power control is implemented easier.

Starvation and high-level deadlocks may still occur. These deadlocks are either programmed (i.e., the computations should terminate anyway) or high-level logical errors. Starvation cannot happen without mutual exclusion. Hence, it is possible, but its possibility can be considerably decreased by a good action scheduler.

The power control has not been mentioned before. Shortly, if a processor finds the queue empty, it can enter a power-saving mode. The scheduler can wake up processors if the queue is longer than the number of active processors.

The proposed subject-oriented programming idea hides the actions of action-oriented models, but does not harm the benefits of action orientation: implicit mutual exclusion and synchronisation, no low-level deadlocks and implicitly parallel execution whenever possible.

Overall, this approach seems to be very attractive especially in environments where computation power and efficient power control is needed. However, although tested in principle, it is still mostly an idea, which requires a lot of further research.

Acknowledgements

I want to thank professors Mikko Tiusanen, Tommi Mikkonen, and Kari Systä for their comments, and especially professor Shmuel Katz from Technion for his inspiring questions that started the way towards subjects.

References

[1] Back, R.J.R., Kurki-Suonio, R., Distributed co-operation with action systems. *ACM Transactions on Programming Languages and Systems 10*, 4 (Oct. 1988), pp. 513–554.

[2] Chandy, K.M., Misra, J., *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[3] Harrison, W., Ossher, H. Subject-oriented programming: a critique of pure objects. *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA '93).* ACM, New York, NY, USA, pp. 411–428. http://doi.acm.org/10.1145/165854.165932

[4] Järvinen, H.-M., Kurki-Suonio, R., DisCo specification language: marriage of actions and objects. *Proc. 11th International Conference on Distribured Computing.* Arlington, Texas, May 1991, IEEE Computer Society Press, 142–151.

[5] Järvinen, H.-M., *The Design of a Specification Language for Reactive Systems.* Doctoral thesis. Tampere University of Technology, 1992. ISBN 951–721–817–6.

[6] Kahneman, D., *Thinking Fast and Slow*, Macmillan, 2011. ISBN 978–1–4299–6935–2.

[7] Kurki-Suonio, R., *A Practical Theory of Reactive Systems*, Springer 2005. ISBN 3–540–23342–3.

[8] Lamport, L., *The temporal logic of actions.* Research Report 71, Digital systems Research Center, 1991.