2014 Second IEEE Working Conference on Software Visualization

Action-Based Visualization

Antti Jääskeläinen, Hannu-Matti Järvinen, and Heikki Virtanen
Department of Pervasive Computing
Tampere University of Technology
Tampere, Finland
Email: {antti.m.jaaskelainen, hannu-matti.jarvinen, heikki.virtanen}@tut.fi

***Abstract*—Many commonly used systems, for example event-based systems, can be considered action systems. Traditionally, action systems have only been visualized behaviorally using state diagrams, which requires translating the system into a state machine and discarding structural information. We introduce an action-based diagram and associated formalism that depict both the behavior and the structure of the system as actions, objects, and participations. The diagrams are useful for visualizing action systems in a native format and can provide an action-oriented viewpoint for other systems as well.**

***Index Terms*—action diagrams; action systems; event-based systems; visualization**

I. Introduction

An *action-based* formalism is a natural way to depict the behavior of event-based systems. The action itself is a computational unit which captures the idea of reactive execution, such as a response to environmental stimulus. An *action system* constitutes a collection of actions operating on variables. The executability of an individual action depends on the values of the variables and its execution may assign new values to them, potentially enabling the execution of new actions. For programming purposes, it is often useful to group variables into objects representing real world entities.

Although the use of action systems is not very widespread, they have been used in specification languages such as Bluespec [1], DisCo [2] and UNITY [3], and model-based testing tools such as fMBT [4], ModelJUnit [5] and OSMO Tester [6]. Event-based systems such as many UIs can also be seen as action systems, and some software design patterns like the DCI architecture [7] show signs of action-based thinking. There are even ideas for implementing action-based computation at hardware level, omitting traditional processes [8].

One impediment to the use of action systems is the lack of a native way to visualize them. Most commonly the behavior of an action system is visualized using state machines, but the required transformation is destructive and all the information on the structure of the system is lost.

Our goal is to address this shortcoming by developing a general visualization method that is genuinely action-based. We present *action diagrams*, which depict an action system as actions, objects and participations. We also formalize the semantics of the diagrams in *General Action Models* (GAM).

We envision twofold benefits from the use of action diagrams. First, any system defined using an action language can be visualized and possibly simulated in its native format, without translating it into a state machine or another completely different formalism. This should make it easier to examine the ideas behind the development of the system. Second, an action-based visualization of any system complements state-based visualization by providing a different viewpoint into its workings, possibly highlighting issues that might easily pass unnoticed in traditional state diagram representations. For example, action diagrams explicitly show the connection between functionality and data, making it easier to see if a variable is always updated when it should be.

Moreover, agile values include communication, simplicity, and feedback. The most valuable illustrations are those available when needed, but nobody has to care about them otherwise. In practice, this means things like hand-drawn sketches and algorithmically generated illustrations. On the other hand, if a diagram is valuable part of persistent documentation, it is useful only if it can be developed and maintained iteratively [9]. The introduction of action diagrams based on formal GAMs makes it possible to generate diagrams automatically from action systems or vice versa with basic compiler techniques.

The rest of the paper is structured as follows: Section II presents a simple example system. In Section III we give an in-depth description of action diagrams and GAMs. Section IV discusses the example and the general features of the action diagrams and their formalism. We review some alternate methods for action-based visualization in Section V and ideas for future development in Section VI. Finally, Section VII concludes the paper.

II. Example

The intended behavior of a simple alarm clock is presented as a state diagram in Figure 1. The picture illustrates nicely what kind of events and when the system is expected to respond to. The action `tick` is triggered by the internal timing mechanism of the clock and marks the passage of time one

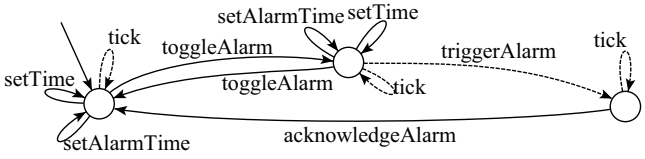

Fig. 1. The behavior of a simple alarm clock as a state machine, with time abstracted away from the state space.

978-0-7695-5305-4/14 $31.00 © 2014 IEEE
DOI 10.1109/VISSOFT.2014.13

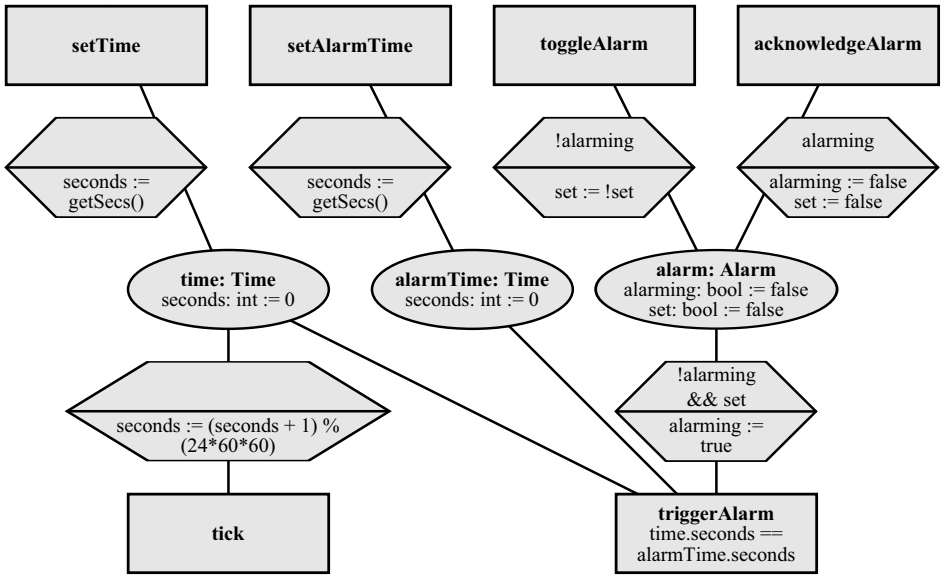

Fig. 2. An action diagram for an alarm clock. The semantics for all types and expressions is that of the C programming language.

second at a time. The actions `setTime`, `setAlarmTime` and `toggleAlarm` are for user interaction. The first two set the current and alarm times of the clock, and the last one sets or unsets the alarm. The action `triggerAlarm` starts alarm at the specified time and `acknowledgeAlarm` is executed when the user switches the alarm off.

Unfortunately, this diagram provides no information about the structure of the implementation nor the variables and their relations. Some of that information is easy to guess in this simple example. There is likely a variable which keeps time and is incremented by `tick`, another for whether the clock is currently alarming, and so on. In a more complex system, these details would be more difficult to infer.

The designer's intentions can be seen explicitly in the action diagram shown in Figure 2. In addition to the actions and bulk state, the diagram shows individual variables and their relation to actions. In the next section, we will explain the contents of diagrams like this in detail.

III. Action diagrams and General Action Models

This section introduces action diagrams and defines their behavior in terms of GAMs. Formal definition and semantics are given for completeness' sake, but the meaning and use of the diagrams should become clear from the informal descriptions.

A. Concepts

An action diagram consists of three kinds of entities: objects, actions and participations. An *object* implicitly contains part of the state of the model in *variables*, which are *names* associated with the object. Variables have *initial values*, which together determine the initial state of the model.

An *action* is the only way to change the state of the model. An action has a set of *preconditions* (also called a *guard*) which determine when it can be executed. The effects of executing an action are determined by its set of *postconditions* (also called a *body*), which consist of assignments to variables within objects.

A *participation* signifies that the variables of an object may be read or written by the action, meaning that the object *participates* in the action, or is a *participant* of the action. The participants of an action must include all objects whose variables are referred to or assigned values in its postconditions (for more on allowing arbitrary references in preconditions, see Section IV). Thus, the execution of an action affects no objects except its participants. The participation itself has no properties in addition to the action and object it is related to.

B. Notation

In an action diagram, an object is marked by an ellipse containing the name of the object and a list of variable definitions (Figure 3). The name of the object is given as *<name> [':' <type>]* and variable definitions follow the format *<name> [':' <type>] ':=' <expression>*, showing the name of the variable, its type, and its initial value. Types may be given to both objects and variables to clarify diagrams based on action languages with typed variables. However, they have no semantic meaning and may be freely left out.

An action is marked with a rectangle containing the name of the action and its precondition, as shown in Figure 4.

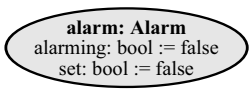

Fig. 3. An object named `alarm`, of type `Alarm`. It contains two variables of type `bool` named `alarming` and `set`, both with an initial value of `false`.

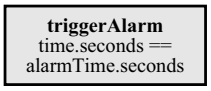

Fig. 4. An action named `triggerAlarm`, with a partial precondition `time.seconds == alarmTime.seconds`. This part of the precondition refers to multiple objects, so it has to be placed here.

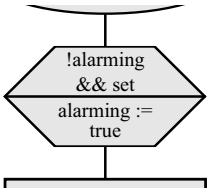

Fig. 5. A participation with a partial precondition `!alarming && set` and an assignment of expression `true` to name `alarming`.

The precondition is written as a truth-valued expression that can refer to any variable, typically with syntax akin to *<object name> '.' <variable name>*. It may be left out if it is identically true. Conjuncts of the precondition that only depend on a single object are given in the participations of objects rather than in the action. The complete precondition is the conjunction of all the partial preconditions. The postcondition of the action is likewise distributed to its participations.

Finally, a participation is marked as a line between an action and a participating object, and an optional hexagon placed on or next to that line (see Figure 5). Pre- and postconditions of an action related to this particular object are placed in the hexagon to avoid cluttering the action rectangles too much. The hexagon has a horizontal line in the middle, with the upper half meant for precondition local to the object and lower half for local postconditions. The local precondition is a conjunct of the action's precondition that refers to this object only. In effect, it shows whether this particular object is ready to participate in the action. The local precondition has the same syntax as in actions, and if identically true, can be left out. The local postconditions contain the assignments to the variables of the object. The syntax for the assignments is *<variable name> ':=' <expression>*. If an assignment is left out, the corresponding variable retains its current value.

If the pre- and postconditions of a hexagon are left out (i.e., the object is only needed in evaluating postconditions or non-local parts of preconditions), the entire hexagon may be left out. Note that a missing hexagon or one with an empty lower half indicates a read-only participant. For convenience, the expressions of the participation may refer to the variables of the participating object directly by their names, with no need to qualify them with the name of the object.

C. Semantics

The behavior of action diagrams is defined in terms of General Action Models (GAM). Formally, a GAM is defined as follows:

Definition 1 (General Action Model). *A General Action Model is a tuple $(O, N, V, A, R, E, I, P, Q)$, where*

- *O is the set of objects*
- *N is the set of names*
- *$V \subseteq O \times N$ is the set of variables*
- *A is the set of actions*
- *$R \subseteq A \times O$ is the set of participations*
- *E is the set of expressions, such that any expression $e \in E$ may refer to variables and can be evaluated into a value*
- *$I : V \longrightarrow E$ is the variable initialization function, such that any expression $I(v)$ refers to no variables*
- *$P : A \longrightarrow E$ is the precondition function, such that any expression $P(a)$ evaluates to a truth value*
- *$Q : \{(a, o, n) \in A \times O \times N \parallel (a, o) \in R \wedge (o, n) \in V\} \longrightarrow E$ is the postcondition function, such that any expression $Q(a, o, n)$ refers only to variables $(\hat{o}, \hat{n}) \in V$ for which $(a, \hat{o}) \in R$*

All expressions of a GAM can always be evaluated into a value. In the case of evaluation errors like a division by zero, the expression evaluates into a special value *undefined*. The semantics of expressions is not defined beyond this in order to support the widest possible range of action languages. In essence, the expressions are treated as black boxes which may refer to variables, and which can be evaluated into a value. The expression semantics should usually be clear from the context, but if not, it should be stated alongside the model or diagram.

All variables are likewise always associated with some value. The state of an individual variable is a pair of that variable and the value associated with it. The state of the whole model is the set of the states of all variables. The initial state of the model is obtained by associating to each variable the initial value designated for it by the initialization function. Formally, if the value associated with the variable $v \in V$ is marked as $val(v)$ and the value evaluated from the expression $e \in E$ is marked as $eval(e)$, then the current state of the model is $\{(v, val(v)) \parallel v \in V\}$ and the initial state is $\{(v, eval(I(v))) \parallel v \in V\}$.

An action can be selected for execution if its precondition evaluates to true. In this situation, the action is considered *enabled*. If multiple actions are enabled, any of them may be executed. If no actions are enabled, the system is in deadlock. Formally, the action $a \in A$ is enabled if and only if $eval(P(a))$. Remember that expressions in the images of the precondition functions evaluate directly to truth values; here an undefined result is treated as false.

When an action is executed, it changes the state of the model according to the assignments of its postcondition. The postcondition follows *single assignment semantics* meaning that references on the right side of the assignment refer to the values before the action execution, and the left side refers to the values after the execution. Thus, assignment to one variable will never affect the evaluation of the expression for another assignment. If any of the expressions evaluate to undefined values, no assignments are performed and the execution of

the action is canceled. Formally, if the value associated with the variable $v \in V$ in the next state is marked as $val'(v)$, then successfully executing the action $a \in A$ in the current state means that $\forall(o,n) \in V : ((a,o) \in R \rightarrow val'(o,n) = eval(Q(a,o,n))) \wedge ((a,o) \notin R \rightarrow val'(o,n) = val(o,n))$.

In a GAM, actions are executed instantaneously and separately, so no rules for parallel execution are required. An actual implementation of an action system may execute actions in parallel, but the results can still be tracked in the GAM as long as no object participates in more than one simultaneously executed action. Likewise, an execution in the model can be performed by an actual system, as long as the actions are scheduled so that the non-instantaneous execution of some actions does not cause other actions to starve (for more on non-instantaneous execution and liveness, see [10, pp. 308–311]). With this caveat, the execution of an action in the model effectively corresponds to finishing its execution in the actual system. Note that a practical implementation of an action system may limit references in preconditions to participating objects only to facilitate their evaluation; a model for such a system must naturally obey the same limitation.

IV. Discussion

Having familiarized ourselves with action diagrams, we can now see how the behavior and structure of the action diagram of Figure 2 correspond to what we saw in and guessed from the state diagram of Figure 1. Notably, there are some differences. For one, the action system allows the setting of time and alarm time while the clock is alarming, because actions `setTime` and `setAlarmTime` do not refer to the variable `alarming` in their preconditions. This is a rather insignificant difference, though, as both behaviors are reasonable.

More significantly, the action diagram shows that the actions `tick` and `triggerAlarm` are completely distinct, such as would be the case if they were implemented in separate processes. This means that it is in principle possible that the trigger does not get to execute during the crucial second. Thus, in addition to the basic liveness assumption (any of the enabled actions shall be selected for execution), a system with this design would require some stronger fairness assumptions.

The state diagram of Figure 1 was a good first approximation for the behavior of the alarm clock, but it did not show the potential for missing an alarm. The possibility could only be shown by adding the time variables into the diagram, and even then might be left out by accident. In contrast, the action diagram with its accurate description of the structure of the system naturally also brings out this quirk of behavior.

The alarm clock example is quite simple and the resulting action diagram is correspondingly small. The diagram for a more complex system could be much larger, and potentially far more difficult to understand. This is not an issue unique to action diagrams; the same problem is encountered with state machine formalisms as well. When modeling complex systems, it is usually necessary to only include some specific aspects of their functionality in a single model, such as a few relevant actions and their participants.

Objects in action diagrams group variables intended to be used together. However, many action languages have only independent variables. In this case, variables can be treated as objects.

GAMs allow the preconditions to refer to arbitrary objects, rather than only the participants of the action. The permissive semantics exists to support action languages with global preconditions, which may depend on all objects. However, some action languages restrict references to participants only.

GAMs have strictly more expressive power than state machines in terms of structure. However, their semantics is quite complex in comparison. Properties such as reachability are liable to be difficult to prove and may indeed be undecidable, depending on expression semantics. As such, GAMs are not very well-suited for the formal analysis of action systems and are likely at their best when used to support the diagrams.

V. Related work

Some alternate methods for action-based visualization exist. Colored Petri nets (CPN) [11] are a very flexible formalism for expressing many kinds of action systems. They are based on places that hold colored tokens, and transitions that consume and produce tokens in places. They can be used very similarly to GAMs, with places corresponding to objects, transitions to actions and tokens of different colors roughly to different object states. The main advantage of GAMs over CPNs is their support for global preconditions, and ability to directly model action languages that have those. Also, GAMs allow variables and their values to be singled out, making the handling of complex objects easier; CPNs can only manipulate aggregate object states as tokens. On the other hand, CPN places can hold an arbitrary number of tokens, which can be useful in modeling data flows. Action languages generally lack such a feature, though.

CPNs are one of the many extensions to the basic Petri net formalism [12], in which tokens are indistinguishable from each other. Ordinary Petri nets have more limited ability to model different action systems, being effectively restricted to handling integer values. However, their simple and elegant semantics lend them well to formal analysis, and their properties have been studied extensively [13].

Event-driven process chains (EPC) [14] depict systems as events that hold the current state, functions that are triggered by and generate events, and logical relationships that connect the first two. Formal semantics has been defined for them based on Petri nets [15]. Overall they are more limited than CPNs, but may be easier to understand.

UML activity diagrams [16] are another option. They are used to express workflows with decisions and concurrent activities. With the liberal use of concurrency, they can be used to model at least some action systems. Nonetheless, they are mostly suited for modeling simpler, fairly linear processes and lack well-defined semantics for complex constructs.

Statecharts [17] might also be relied on in a pinch. They extend state machine formalism into parallel substates. However, despite modeling parallelism, they are still essentially

state diagrams, and their ability to depict systems in an action-based format is very limited.

The composition of systems in process algebras is sometimes illustrated with connection diagrams [18, pp. 54–55]. These diagrams show processes and symbols which serve as the points of communication in them; this representation is very similar to the relation between objects and actions. Connection diagrams by themselves cannot be used to describe the behavior of a system, though.

Use case maps [19] likewise show the interaction between functionality and data, in this case as use cases marked into a class diagram. Use case maps can be used to represent individual execution traces and their relation to program structure, but they cannot display the entire behavior of the system.

VI. Development Ideas

GAMs and action diagrams depict the system at instance level, as objects and actions. A type level view with classes and action types instead of objects and actions could also be used. Such a diagram would have the advantage of condensing the visualizations of some systems into smaller space. Moreover, type level diagrams might be well-suited for visualizing properties that are hard to see from instance level diagrams.

There are also some new features that could be added to the diagrams. One is the ability to depict dynamic systems, where objects and/or actions can be created and destroyed during execution. At the moment, this requires auxiliary information to keep track of whether entities are supposed to exist and works only when their numbers have strict upper limits.

Another useful feature is *triggered actions*: actions which are executed only once when they become enabled, and will not be executed again unless the state of at least one of the participants changes. They can be simulated by state variables, but this is unnecessarily cumbersome.

The current model also lacks proper timing features. As the example above shows, time may be simulated with an explicit action devoted to it, but that is hardly practical for systems with complex real-time requirements. As such, a method for setting timers and deadlines could prove useful.

Action diagrams can be sketched with paper scraps and whiteboard, or created with general diagram drawing tools. However, a dedicated tool would make drawing the diagrams easier and could verify their syntactic correctness. Diagrams could also be automatically generated from action systems and vice versa. In addition, tool support for simulating execution in the diagrams would greatly ease their informal analysis, although any simulation tool would likely have to be limited to some specific expression semantics.

VII. Conclusion

Action diagrams can be used to represent systems in a format that is intuitively action-based. The diagrams consist of objects that hold data, actions that manipulate data, and participations that connect objects with actions.

The diagrams are formalized as General Action Models (GAM), which ensures that their semantics is well-defined. The model formalism is designed to be general in nature so that action systems created in different action languages can be easily expressed in its terms.

Current GAMs are usable for representing the basic properties of action systems. Planned new features include support for dynamic or timed systems, and diagrams for visualizing systems at type level. In any case, tool support for diagram creation and simulation should be developed.

More work is needed to evaluate the benefits of action diagrams in practice. Case studies need to be performed in different domains and for different action languages.

References

[1] Bluespec, Inc., "Bluespec™ system Verilog reference guide," http://www.ece.ucsb.edu/its/bluespec/doc/BSV/reference-guide.pdf, 2008, cited Aug. 2014.

[2] H.-M. Järvinen and R. Kurki-Suonio, "DisCo specification language: Marriage of actions and objects," in *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS 1991)*. Los Alamitos, CA, USA: IEEE Computer Society, May 1991, pp. 142–151.

[3] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Reading, Massachusetts: Addison-Wesley, Nov. 1988.

[4] Intel Corporation, "fMBT | 01.org," https://01.org/fmbt/, 2014, cited Aug. 2014.

[5] M. Utting, G. Perrone, J. Winchester, S. Thompson, R. Yang, and P. Douangsavanh, "The ModelJUnit model-based testing tool," http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/, 2009, cited Aug. 2014.

[6] osmo – A model-based testing tool, http://code.google.com/p/osmo/, 2014, cited Aug. 2014.

[7] T. Reenskaug and J. O. Coplien, "The DCI architecture: A new vision of object-oriented programming," http://www.artima.com/articles/dci_vision.html, 2009, cited Aug. 2014.

[8] H.-M. Järvinen, "Actions, objects, and subjects," in *Proceedings of the 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'13)*, H. R. Arabnia, H. Ishii, M. Ito, K. Joe, H. Nishikawa, and F. G. Tinetti, Eds., vol. I. Athens, GA, USA: CSREA Press, Jul. 2013, pp. 285–291.

[9] S. W. Ambler, *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. Wiley, Mar. 2002.

[10] R. Kurki-Suonio, *A Practical Theory of Reactive Systems*, ser. Texts in Theoretical Computer Science. Berlin, Heidelberg: Springer, 2005, vol. XXII.

[11] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*, 2nd ed. Berlin, Heidelberg: Springer, Jul. 2009.

[12] C. A. Petri, "Kommunikation mit automaten [Communication with automata]," Ph.D. dissertation, University of Bonn, Bonn, Germany, Jun. 1962, in German.

[13] J. Esparza and M. Nielsen, "Decidability issues for Petri nets - a survey," *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, vol. 52, pp. 244–262, Feb. 1994.

[14] ARIS Community, "Event-driven process chain (EPC)," http://www.ariscommunity.com/event-driven-process-chain, 2014, cited Aug. 2014.

[15] K. van Hee, O. Oanea, and N. Sidorova, "Colored Petri nets to verify extended Event-driven Process Chains," in *Proceedings of the 4th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS'06)*, J. Barjis, U. Ultes-Nitsche, and J. C. Augusto, Eds. INSTICC Press, May 2006, pp. 76–85.

[16] Object Management Group, Inc., "Unified Modeling Language (UML)," http://www.omg.org/spec/UML/Current, 2014, cited Aug. 2014.

[17] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[18] C. A. R. Hoare, "Communicating sequential processes," Jun. 2004, available at http://www.usingcsp.com/cspbook.pdf, cited Aug. 2014.

[19] R. J. A. Buhr and R. S. Casselman, *Use Case Maps for Object-Oriented Systems*. Prentice Hall College Div, 1996.