Creation of DEVS Models using Imitation Learning

Michael W. Floyd and Gabriel A. Wainer
Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive, Ottawa, ON, Canada
{mfloyd, gwainer}@sce.carleton.ca

Keywords: DEVS, imitation learning, case-based reasoning, real-time, transfer learning

Abstract

Modelling and simulation of robotic control systems allows for low cost analysis and experimentation. However, creating these models requires a level of technical expertise. To improve the technical quality of such models, we propose a case-based reasoning approach to learn the behaviour of models using the DEVS formalism. By observing a desired behaviour, in the form of outputs produced in response to inputs, a DEVS model of the behaviour can be built. This model can then be used during simulation to imitate the behaviour of interest. Our results show that this learning approach can be used to successfully imitate the behaviour of several DEVS models. Additionally, it can be used to transfer behaviour to and from a non-DEVS model.

1. INTRODUCTION

Modelling and simulation (M&S) allows for analysis and experimentation on a variety of systems. One class of systems M&S can be used on is robotic control. These systems are often modelled based on human-defined behaviour so the robot reacts to its environment in a specific way as defined by an expert. However, developing such a model requires some level of technical skill. This could be knowledge of the modelling formalism being used or of the programming language used to implement the model. In order to make easier the work of a domain expert we look to *learn* the desired model behaviour.

The ability to learn a model's behaviour is beneficial in three primary ways. First, as mentioned previously, it allows the transfer of knowledge from a non-technical user into a model. This simplifies the user's technical skill requirements. Secondly, even if the user is technically able to create the model they may not wish to explicitly create the model, possibly due to time constraints. Instead they could demonstrate the desired behaviour which would require significantly less of their time. Lastly, it would be possible to learn the behaviour of other models. This would be particularly useful if the model was created with a different formalism or incompatible technology. The model behaviour could then be converted for use in an existing modelling framework.

In order to perform this learning, we first model the human or model as a black box. No assumptions are made about the underlying design or behaviour of the black box, only that it receives some external inputs and produces outputs in response (Figure 1). We can say that the output events, E_o, are based on the input events, E_i, as a function of the internal reasoning process of the black box:

$$E_o = f(E_i) \tag{1}$$

While we may not know the details of this function, we can attempt to approximate it based on observations of the inputs and outputs. After collecting a series of such observations, it then becomes possible to learn how the black box will behave when it receives input. Thus, we can then create a model that uses this information to imitate the behaviour of the black box.

Figure 1. Representing the human or model as a black box

The remainder of this paper will detail a method for learning the behaviour of a discrete-event system specification (DEVS) model using imitation learning. Section 2 provides background information about DEVS, embedded CD++, case-based reasoning and related work. The DEVS models used to observe and imitate are presented in Sections 3 and 4. Experimental results using these models are described in Section 5 followed by conclusions in Sections 6.

2. BACKGROUND

Discrete-event system specification (DEVS) [1] is a formal framework for modelling and simulation. The beneficial properties of the DEVS formalism include allowing for individual atomic models to to be coupled into hierarchies and modules, support for discrete-event approximations of continuous systems, and the ability to separate the model from the simulator. An atomic DEVS model (AM) is formally represented by the following tuple:

$$AM = (X, Y, S, ta, \delta_{ext}, \delta_{int}, \lambda) \quad (2)$$

The model is able to receive input events from the set of possible inputs, X, which causes its external transition function, δ_{ext}, to be invoked. This function may change the state of the model, to a state in S, and set the time for that state using the time advance function, ta. If the state remains unchanged for the duration of time set by ta, the internal transition function, δ_{int}, is called and can modify the model state and set the associated time advance. Following the internal transition function the output function, λ, is called and may produce an output event from the set of possible outputs, Y.

Embedded CD++ (eCD++) [2] is an extension of the CD++ toolkit [3] that allows for real-time execution of models. Both CD++ and eCD++ allow models to be defined using the DEVS formalism. The primary advantage of these tools is that they provide a clear separation between the models and simulators. This allows models to be created without any knowledge of the underlying simulation engine, so the same model can be used with a variety of simulators.

The standard version of CD++ operates in simulated time, so the internal simulator clock advances based on the time of the next scheduled event. Embedded CD++, however, allows for simulation in real-time so events are processed at their scheduled time. Since models can be simulated in real-time, they are able to interact with external hardware. These hardware devices can produce input events for the DEVS model or receive output events (Figure 2). If the simulation was not in real-time, the internal simulation time might differ from the real-world time causing synchronization problems when hardware events are received.

Figure 2. Using embedded CD++ to interact with external hardware

Models can then be used at various stages of the modelling and simulation process using eCD++. Initial simulation can be done in simulated-time using simulated input events. Provided the model's behaviour can be verified and validated in simulated-time, the same model can then be used in real-time to interact with actual hardware. This allows a more thorough testing of the model since interacting with hardware might produce events that were not considered during simulated-time experimentation.

Case-based reasoning (CBR) [4] is a learning method that relies on the assumption that *similar problems* have *similar solutions*. CBR makes use of instances of previously encountered problem-solution pairs, called *cases*, that are used to solve novel input problems. In CBR, a collection of cases, called a *case base*, is used to represent the knowledge learnt by the system. When an input problem is received, a CBR system compares the input to cases in the case base. Cases that have similar problems as the input problem can then have their solutions used to determine how the input problem should be solved.

Case-based reasoning lends itself well to a task like imitation learning because ideally the imitator should behave exactly as the teacher would when presented with the same input problems. In the case of a DEVS model, the problem would be the inputs received and the solution would be the outputs produced. In order to successfully imitate the model's behaviour it becomes necessary to observe which inputs cause the model to produce which outputs, and in turn to use those observed cases to determine how to behave during run-time.

Learning by imitation and observation has been utilized in a variety of domains, both physical and simulated. Many early robotic applications looked at learning simple movement tasks for robotic arms [5] but have since moved to more advanced tasks like controlling aerial maneuvers performed by robotic helicopters [6]. In these robotic domains the learning algorithms have prior knowledge of the dynamics of the individual robot and learn the various control parameters. The learning is generally tightly coupled to the specific task being performed by the robot so it is not directly applicable to general learning tasks.

There are many promising uses of case-based reasoning for imitation learning in games. Examples of this include Tetris [7], real-time strategy games [8], poker [9] and space invaders [10]. The primary limitation of these works is that they are domain specific and require information about the tasks being imitated. This information can include knowledge about the goals of the task or utilizing data structures and algorithms that are specific to a certain domain. The requirement for information about the task being performed is not limited to CBR as other approaches to imitation learning suffer similarly [11].

In our recent work we have focused on a domain-independent approach to software agent imitation using case-based reasoning [12, 13, 14]. However, we have only utilized ad hoc simulators and never formal modelling techniques like DEVS. One of the primary contributions of this work is the ability to imitate formal models while the imitator itself adheres to a modelling formalism. Additionally, this work differs from previous works in that it allows the transfer of learnt behaviour from robotic to simulated domains and from formal to ad hoc models (and the other way around).

3. MODEL OBSERVATION

In order to imitate the behaviour of a DEVS model, it first becomes necessary to *observe* how the model responds to input events[1]. This observation is performed with a model, *Observer*, that is placed between the DEVS model being observed and the external hardware (Figure 3). The Observer model acts as an intermediary between the DEVS model and the external hardware, recording any events that pass through it.

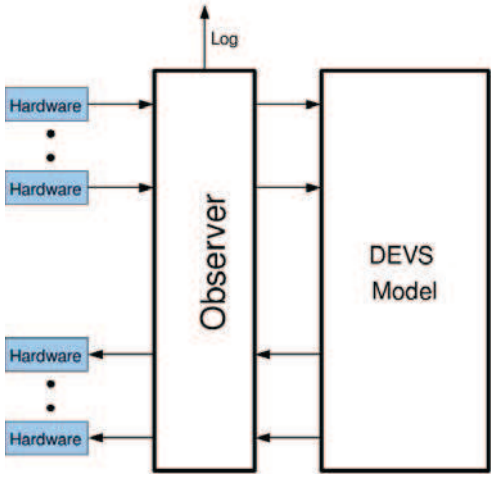

Figure 3. Using the Observer model to log the behaviour of a DEVS model

More formally, the Observer can be defined as a DEVS atomic model:

- **X:** There is one input port for each input and output port of the model being observed. The events sent on these ports are not known in advance and are dependant on what events can be sent or received by the observed model.
- **Y:** Similarly, there will be one output port for each input and output port of the observed model. Each output port will be paired with an input port. There will also be one extra output port, shown as Log in Figure 3, that outputs the observations.
- **S:** The model is either in an active or passive state. The model has a number of state variables. Three variables to record the last event received from external hardware, the time the event was received and the port it was received on. Three similar variable are used for events from the observed model as well. Also, a variable is used to record which port received the last event (can be either from external hardware or observed model).
- **ta:** A short time advance is used to simulate the time it takes to record the observations.
- δ_{ext}**:** An external event means that an event was received from the external hardware or the observed model. Whichever port received the input will have its associated state variable updated with the event value and the time of the event. State variables will also be updated to identify which port received the last event. The model will then enter an active state.
- δ_{int}**:** The model will return to a passive state and wait for further input.
- λ**:** The last received value will be sent as output on the appropriate output port. If the last received event from the observed model, a message will be sent on the Log port that contains the information contained in the state variables.

This model will record all incoming events before transmitting them along. As the model executes, it will create a log of output events produced by the observed model and the input event that the were received previously. These logged observations will be used to create cases that will be used during imitation.

As was previously discussed, in case-based reasoning a case, C, contains a problem-solution pair:

$$C = (P,S) \tag{3}$$

We will define the problem, P, to represent the input events received by the model. The problem can then be decomposed into the port, p, where the input was received and the value, v, of the event:

$$P = (p,v) \tag{4}$$

The solution, S, is then the output events produced by the observed model. Each solution is a series of output events:

$$S = [s_1, s_2, \ldots, s_n] \tag{5}$$

And each individual output event, s_i, is a triple composed of the event port, p, event value, v, and amount of time, t, the event takes.

$$s_i = (p,v,t) \tag{6}$$

There are two possible options for observation and case generation: passive and active. In passive observation, the DEVS model receives events during the normal course of execution. For example, the model could be receiving actual events while operating in real-time. The other option is active observation, where the input events are artificially generated. While a passive approach may result in a more realistic series of input events, it gives less control over which events are encountered. If a model was deployed in a robot, there would

[1]For the remainder of this paper we refer to the imitation of DEVS models, however, the same principles apply to the imitation of human behaviour. Any parts of the learning system that interact with the DEVS model could instead connect to an interface controlled by a human.

be no guarantee that a representative sample of input events would be seen. Due to these limitations, we will utilize an active observation approach (Figure 4).

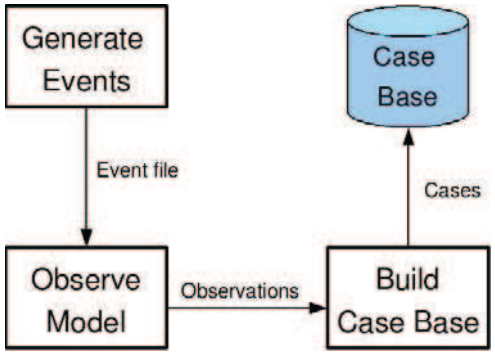

Figure 4. The process used to observe a model and build a case base. The Observe Model process would be performed by the Observer model from Figure 3

Input events are automatically created by randomly selecting an input port and then randomly generating a value to send on that port. A series of such events can be generated and stored in an *event file* which is then used as input to the coupled model containing the Observer model and the model being observed (Figure 3). As input events are processed by the Observer model, it will log observations which can be converted to cases. Once a number of cases have been stored in the case base it then becomes possible to imitate the model.

4. MODEL IMITATION

The cases that are generated by observing a DEVS model can be used to imitate the model's behaviour. To accomplish this another DEVS model, called the *imitation model*, is used in place of the model being imitated (Figure 5). When the imitation model receives inputs, it attempts to behave as the original model would have.

Figure 5. Using the Imitation model in place of the original model

The inputs are treated as novel problems without a known solution. To find the solution, a k-nearest neighbour search is used[2]. In a k-nearest neighbour search a novel input problem is compared to known problem instances, in this situation cases in the case base, and the k most similar instances are found. These k problem instances, which already have known solutions, can then be used to determine the solution to the input problem. This requires some way to measure how similar two cases are. Based on the problem definition from the previous section, we define the distance between two problems as:

$$distance(P_i, P_j) = \begin{cases} |v_i - v_j| & ,if\, p_i = p_j \\ \infty & ,if\, p_i \neq p_j \end{cases} \quad (7)$$

Therefore, the k-nearest neighbour search finds the cases that are the minimum distance from the input problem. We utilize a 1-nearest neighbour search so the solution portion of the closest case is used as the solution to the input problem and the events found in that solution are sent as output.

The DEVS specification for the imitation model has the same set of input events, X, and output events, Y, as the model it is imitating. The model can either be passive or active. It keeps state variables containing the next event value to send, the port to send it on, and a list of any other events that need to be sent. When an external input event is received, the external transition function, δ_{ext}, performs a 1-nearest neighbour search to find the best solution to perform. The state variables are then updated to reflect the solution that will be performed. The time advance, ta, depends on each event. The imitation model attempts to keep event durations the same as they were in the original model. When the internal transition function, δ_{int}, is invoked, if there are more events to send the next is set to be sent. Otherwise, the model becomes passive. The output function, λ, then sends the next event value on the appropriate output port.

The advantage of this approach to imitation is that no assumptions are made about the model being imitated. It is not necessary to modify the imitator model or add any domain knowledge in order to imitate different behaviours. Instead, only the case base needs to be changed. Several case bases could be created, one for each DEVS model being imitated, and simply interchanged depending on the desired behaviour. This is particularly useful if a novel behaviour is required under tight time constraints. The behaviour could be demonstrated while being observed by the Observer model (from the previous section), a case base could be created and used directly by the imitation model.

5. SIMULATION AND RESULTS

In order to demonstrate the use of our imitation approach we will discuss the following experiments:

- Imitation of an embedded CD++ model of an obstacle avoidance robot.
- Imitation of an embedded CD++ model of a robotic arm.

[2]Previous work [12, 13, 14] has shown imitation learning systems using case-based reasoning with a k-nearest neighbour search to perform with a high degree of accuracy. This has been shown both quantitatively and qualitatively by observing the behaviour of the agent.

- Transferring behaviour from a RoboCup [15] soccer agent to a DEVS model.
- Transferring learnt behaviour to a RoboCup soccer agent.

The results from these experiments will show not only that imitating DEVS models is possible, but that it can be done with a high degree of accuracy. In fact, we will show that it is difficult to differentiate the original model from the imitator.

5.1. Obstacle Avoidance Model

In this experiment, we look to imitate the behaviour of an existing DEVS model that performs obstacle avoidance in a mobile robot called RoboCart [2]. This model receives input events from two external hardware sensors, touch and sonar, and can send events to motors that move the robot. The touch sensor produces an event when it touches an object, whereas the sonar sensor periodically produces events containing the distance to the nearest object. The model produces events that move the robot forward (the value 1), backward (the value 2), turn left (the value 3) or turn right (the value 4). The model causes the robot to move forward until it bumps into an object (receives input from the touch sensor) or detects an object nearby (receives input from the sonar sensor).

An event file was generated containing 500 randomly selected input events. The obstacle avoidance model was coupled with the Observer model (as described in Section 3) and the event file was used as input, which produced 500 cases. These cases were then used by the imitation model.

In order to test the ability of the imitation model to reproduce the behaviour of the original model, a second event file was randomly generated. This event file was used as input to both the imitation model and the original and their outputs were compared. A smaller example of such an event file is shown in Figure 6 with the imitation results in Figure 7 and the original model's results shown in Figure 8. This event file represents the following events:

1. The robot touched an object
2. An object was sensed at a distance of 11.72
3. An object was sensed at a distance of 3.81
4. The robot touched an object
5. An object was sensed at a distance of 16.55

The first thing to notice when examining these results is that they both contain the same number of events and those events occur at identical times. This shows that not only is the imitation model able to successfully determine when to send output events but it is also able to determine the duration of those events. In general, the values of the events are identical as well. There are several situations where the event values are not identical and they all relate to the turn direction (value 3 or 4). This is because the original model does not use input events to determine which direction to turn, it simply toggles between turning left and right. The output for the imitation model can be interpreted as follows:

```
00:00:00:000 intouch 1000
00:00:10:000 insonar 11.72
00:00:20:000 insonar 3.81
00:00:30:000 intouch 1000
00:00:40:000 insonar 16.55
```

Figure 6. Sample input to test the imitation of the obstacle avoidance model

```
00:00:00:020 out 2
00:00:01:040 out 3
00:00:02:560 out 1
00:00:10:020 out 3
00:00:11:540 out 1
00:00:20:020 out 2
00:00:21:040 out 3
00:00:22:560 out 1
00:00:30:020 out 2
00:00:31:040 out 3
00:00:32:560 out 1
00:00:40:020 out 4
00:00:41:540 out 1
```

Figure 7. Output events produced by the imitation model

1. In response to touching an object: the robot is moved in reverse, turns left and then goes forward again
2. In response to sensing an object at medium distance: the robot is moved left and then forward
3. In response to sensing an object at close distance: the robot is moved in reverse, left and then forward
4. In response to touching an object: the robot is moved in reverse, turns left and then goes forward again
5. In response to sensing an object at medium distance: the robot is moved right and then forward

However, even though it does not always turn in a manner that is identical to the original model the imitation model is still able to behave in a very similar way. When the imitation model is deployed in the robot, in real-time eCD++ simulation mode, it is difficult to tell that the robot is not being controlled by the original model. The only way to notice the difference is if you are aware that the original model never causes the robot to turn in the same direction twice in a row. The imitation model clearly displays the obstacle avoidance behaviour we would expect from the original model.

```
00:00:00:020 out 2
00:00:01:040 out 3
00:00:02:560 out 1
00:00:10:020 out 4
00:00:11:540 out 1
00:00:20:020 out 2
00:00:21:040 out 3
00:00:22:560 out 1
00:00:30:020 out 2
00:00:31:040 out 4
00:00:32:560 out 1
00:00:40:020 out 3
00:00:41:540 out 1
```

Figure 8. Output events produced by the original model

5.2. Robotic Arm Model

We now look to further experiment on the imitation model by learning from the model for a robotic arm. The robotic arm has three input ports that receive input events from a sound, touch and light sensor. It can send events on two output ports: one that controls the arm and one for the gripper claw. When the model receives an event from the sound sensor (indicating a significantly loud sound) the arm begins moving. It continues to move until it receives an input from the touch sensor indicating it has touched something. The light sensor, which has a short range and needs to be next to an object to successfully determine colour, is then able to produce an event indicating what colour the object is that was touched. If the object was red, the gripper claw closes on the object and the arm moves in reverse. If the object was blue, the arm moves in reverse without gripping the object.

As with the obstacle avoidance model, a random event file of 500 events is created. However, this model does not run continuously. It runs one cycle of its behaviour (move to an object, examine it and move back) and then terminates. This means that it is impossible to watch the model grab a red object and leave a blue object in a single observation session. To overcome this 10 event files are randomly generated and used during 10 observation sessions. The cases from each of these observation sessions are then combined into a single case base.

We look to test the two primary behaviours of the robotic arm. Figure 9 shows a sample input event file that represents encountering a red object and Figure 10 for a blue object. The events in these files represent:

1. The robot heard a sound with volume 35
2. The robot touched an object
3. The robot sensed the object was red (when the value is 5) or blue (when the value is 15)

```
00:00:03:00 insound 35
00:00:05:00 intouch 1000
00:00:06:00 inlight 5
```

Figure 9. Sample input to test the imitation of grabbing a red object

```
00:00:03:00 insound 35
00:00:05:00 intouch 1000
00:00:06:00 inlight 15
```

Figure 10. Sample input to test the imitation of not grabbing a blue object

Unlike with the obstacle avoidance imitation, where there were minor differences, both the original model and imitating model produce identical output for the robotic arm. The output when simulating a red object is shown in Figure 11 and for a blue object in Figure 12. These outputs can be interpreted as follows:

1. Sounds heard: The arm begins moving forward (value of 1)
2. Object touched: The arm stops (value of 2)
3. Colour sensed: If the object was red the claw is closed (value of 3) and then stopped (value of 6). Regardless of colour, the arm is moved in reverse (value of 5) and then stopped (value of 2).

```
00:00:03:020 outarm 1
00:00:05:020 outarm 2
00:00:06:030 outclaw 3
00:00:08:030 outclaw 6
00:00:09:020 outarm 5
00:00:10:040 outarm 2
```

Figure 11. Output when simulating the robotic arm encountering a red object

These experiments show that this imitation approach can be used for a variety of models and is not restricted to imitating a specific behaviour. The obstacle avoidance model and the robotic arm model behave in a significantly different manner and make use of different external hardware. The imitation model itself does not have to be changed to imitate these different behaviours, instead only the case base has to be changed.

5.3. Transferring Behaviour from a Non-DEVS Simulator

Up to this point the focus has been exclusively on imitating the behaviour of DEVS models. Another potential use of

00:00:03:020 outarm 1
00:00:05:020 outarm 2
00:00:06:020 outarm 5
00:00:07:040 outarm 2

Figure 12. Output when simulating the robotic arm encountering a blue object

imitation learning is to transfer behaviour to different simulators. This would be useful if the majority of models were implemented using a specific formalism but some of the models used a different formalism. The behaviour of these other models could be learnt and the behaviour could be imitated by a model implemented in the desired formalism.

In these experiments we look to transfer the behaviour from a non-DEVS simulator into a DEVS model. We make use of the RoboCup soccer simulator. RoboCup is a popular simulation platform that allows software agents to compete in games of simulated soccer[3]. Numerous international competitions occur each year allowing researchers to compare various artificial intelligence techniques in a friendly competition.

Simulated RoboCup soccer operates in a client-server architecture. The central server handles simulation, enforces the rules of soccer and coordinates messages to the various clients (each of which represents a single soccer player). The messages sent from the server to the players contain information about the objects the player can currently see in its field of vision and the distance those objects are from the player. The objects a player can see include soccer balls, boundary lines, boundary flags, goal nets and other soccer players. In response the player can send a message to the server indicating what action they want to perform (like kicking the ball or moving around the field).

The RoboCup agent we attempt to learn from performs an object tracking behaviour. If it is unable to see the soccer ball in its field of vision it turns until it can see the ball. When the ball is visible, it moves toward the ball. In order to observe this agent and build a case base we use an approach similar to the observer model (in Section 3) but with a modified case structure (a detailed description is provided in [12]). The problem that arises is that the cases in the RoboCup domain are at a higher level of abstraction and contain information about objects that are visible to the agent instead of sensor inputs. This representation is incompatible with both the case structure we have defined and also the sensory capabilities of the robot we will deploy the model in.

In order to transfer the RoboCup cases into the cases we have defined, we need to perform a conversion. We use the following mapping which converts RoboCup cases to cases usable by the RoboCart robot:

Problem Mapping:

- The ball is visible and directly infront of the agent in the RoboCup case → An event on the sonar port with the distance of the object
- Otherwise → An event on the sonar port that represents no object is visible

Solution Mapping:

- A RoboCup *dash* action → A forward event on the motor control port.
- A RoboCup left *turn* action → A left turn action on the motor port.
- A RoboCup right *turn* action → A right turn action on the motor port.

Duration Mapping:

- All events are given a similar duration (300 ms).

This mapping helps convert RoboCup cases to RoboCart cases but there is some information loss. Since the RoboCart does not have the necessary sensors to uniquely identify objects it does not only track soccer balls but tracks any objects it senses on its sonar sensor. Even with this limitation, during real-time execution the robot successfully imitates the behaviour that was learnt from the RoboCup agent. This experiment helps show that it is possible for behaviour programmed in non-DEVS modelling formalisms to be successfully converted to DEVS models. These models can then be integrated with DEVS models so that all models use the same simulation engine.

5.4. Transferring Behaviour to a Non-DEVS Simulator

In the previous subsection we looked at transferring the behaviour of a non-DEVS model to a DEVS model, but now we will look to do the reverse. We will transfer the behaviour that was learnt while observing the obstacle avoidance robot to a RoboCup agent. As was the case previously, a method is required to map cases:

Problem Mapping:

- Input event on the touch sensor port → A RoboCup ball object at a small distance from the agent
- Input event on the sonar sensor port → A RoboCup ball object at a distance equal to the input event value

Solution Mapping:

- Move the robot forward → A RoboCup *dash* action

[3]Robotic competitions also exist but we will focus exclusively on the simulated league.

- Move the robot backward → A RoboCup backward *dash* action
- Move the robot left → A RoboCup left *turn* action
- Move the robot right → A RoboCup right *turn* action

In this mapping ball objects were used as obstacles, but this could be changed to any type of RoboCup object.

Using a case base created with this mapping, obstacle avoidance behaviour can be successfully transferred to a RoboCup agent. The agent clearly avoids colliding with the ball and behaves very similarly to the obstacle avoidance robot. This shows that imitation learning can be used to transfer behaviour in both directions: to DEVS models and from DEVS models. This makes models more portable as they can quickly be transferred to a different modelling framework without having to reimplement the model. Additionally, interoperability is promoted since models can be converted even if their behaviour is not fully known or if their source code is unavailable.

6. CONCLUSIONS

In this paper we have demonstrated an approach to developing DEVS models using imitation learning. Rather than explicitly implementing a model, the model behaviour is learnt though observation. While this approach removes the need for technical skills that are typically required for knowledge transfer, it is not appropriate for all situations. Firstly, this approach involves *learning* so it may not provide an exact imitation of the desired behaviour. This becomes an issue if the model is to be deployed in a situation that is not tolerant of error. Secondly, the learning approach assumes that the model outputs are a function of the inputs. If a model does not use recent inputs to generate outputs then this approach would be unable to successfully imitate the model's behaviour. Lastly, if a teacher never performs a certain behaviour or action while being observed then the imitator will never be able to perform that behaviour.

Even with the listed limitations, this approach still provides a novel technique for developing DEVS models. The two primary contributions of this work are demonstrating the ability of imitation learning to learn the behaviour of a DEVS model and showing how behaviours can be transferred to and from non-DEVS models. In our experiments we show how two robotic control models, one for an obstacle avoidance robot and one for a robotic arm, can be successfully learnt. We also show that behaviour programmed in a simulator that does not follow a modelling formalism, simulated Robocup soccer, can be converted to or from a DEVS model.

REFERENCES

[1] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2000.

[2] M. Moallemi, J. M. Gutierrez-Alcaraz, and G. A. Wainer, "ECD++ a DEVS based real-time simulator for embedded systems," in *Spring Simulation Multiconference*, p. 12, 2008.

[3] G. A. Wainer, "CD++: A toolkit to define discrete-event models," *Software, Practice and Experience*, vol. 32, no. 3, pp. 1261–1306, 2002.

[4] A. Aamodt and E. Plaza, "Case-based reasoning: Foundational issues, methodological variations and system approaches," *AI Communications*, vol. 7, no. 1, pp. 39–59, 1994.

[5] C. G. Atkeson and S. Schaal, "Robot learning from demonstration," in *Fourteenth International Conference on Machine Learning*, pp. 12–20, 1997.

[6] A. Coates, P. Abbeel, and A. Y. Ng, "Learning for control from multiple demonstrations," in *25th International Conference on Machine Learning*, pp. 144–151, 2008.

[7] H. Romdhane and L. Lamontagne, "Reinforcement of local pattern cases for playing Tetris," in *21st International Florida Artificial Intelligence Research Society Conference*, pp. 263–268, 2008.

[8] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *7th International Conference on Case-Based Reasoning*, pp. 164–178, 2007.

[9] J. Rubin and I. Watson, "SARTRE: System overview. A case-based agent for two-player texas hold'em," in *Workshop on CBR for Computer Games at the 8th International Conference on Case-Based Reasoning*, 2009.

[10] M. Fagan and P. Cunningham, "Case-based plan recognition in computer games," in *5th International Conference on Case-Based Reasoning*, pp. 161–170, 2003.

[11] C. Thurau and C. Bauckhage, "Combining self organizing maps and multilayer perceptrons to learn bot-behavior for a commercial game," in *GAME-ON Conference*, 2003.

[12] M. W. Floyd, B. Esfandiari, and K. Lam, "A case-based reasoning approach to imitating RoboCup players," in *21st International Florida Artificial Intelligence Research Society Conference*, pp. 251–256, 2008.

[13] M. W. Floyd, A. Davoust, and B. Esfandiari, "Considerations for real-time spatially-aware case-based reasoning: A case study in robotic soccer imitation," in *9th European Conference on Case-Based Reasoning*, pp. 195–209, 2008.

[14] M. W. Floyd and B. Esfandiari, "An active approach to automatic case generation," in *8th International Conference on Case-Based Reasoning*, pp. 150–164, 2009.

[15] RoboCup, "Robocup official site." http://www.robocup.org, 2009.