A JavaPathfinder Extension to Analyse Human-Machine Interactions

Sébastien Combéfis*, Dimitra Giannakopoulou†, Charles Pecheur* and Peter Mehlitz†
*Computer Science and Engineering Department
ICT, Electronics and Applied Mathematics Institute
Université catholique de Louvain, Louvain-la-Neuve, Belgium
Email: {Sebastien.Combefis,Charles.Pecheur}@uclouvain.be
†NASA Ames Research Center
Moffett Field, CA 94035, USA
Email: {Dimitra.Giannakopoulou,Peter.C.Mehlitz}@nasa.gov

***Abstract*—We present `jpf-hmi`, a Java Pathfinder (JPF) extension that supports the description and analysis of human machine interaction (HMI) systems. The extension is built on top of `jpf-statechart`, but differentiates between events in terms of commands, observations and internal actions, as it is typical in the HMI domain. `jpf-hmi` implements two algorithms for generating concise system models for human operators. It also supports the detection of several types of HMI-specific anomalies known as "automation surprises", such as non full-control determinism and mode confusion. These capabilities are provided in addition to the existing more generic property verification that is supported by JPF, and which can also be applied to HMI systems.**

***Keywords*-JavaPathfinder extension; human-machine interaction; formal methods; modelling; bisimulation; learning**

I. Introduction

Interaction between human operators and automated systems is getting increasingly complex and error-prone, resulting in many accidents or system failures due to automation surprises [1]–[3]. Formal methods can be useful for performing rigorous analysis of human-machine interactions.

Recently, Heymann and Degani [4] developed a method to automatically generate so called "mental models", which are minimal abstractions of HMI systems that enable operators to control such systems solely based on knowledge of the corresponding mental model. Their approach was later studied more extensively resulting in two algorithms for automatically generating such abstraction: the first is based on reduction [5] and the second on learning [6].

This paper presents the `jpf-hmi` prototype, an extension of the Java Pathfinder model checker (JPF) [7] that provides the capability to analyze human-machine interactions based on techniques developed in our previous work [5], [6], [8]. The tool allows the designer to input HMI models, which are analyzed by `jpf-hmi`.

The remainder of the paper is organized as follows. Section II states the necessary background to understand the analysis techniques. Section III presents the extension, its architecture and an example illustrating how to use it. Finally, the last section concludes the paper by presenting future work.

II. Interaction Modelling and Analysis

This section provides a quick overview of the background necessary to understand the tool presented in this paper. The detailed formalization and algorithm description is presented in [5], [6], [8]. We use a simple running example of a countdown system. The countdown starts at 4 with the `_start()` command. The value then decreases by steps of 1, with an observable `_tick()` action. Finally, the user can reset the countdown, any time after the first tick, with the `_reset()` command. Figure 1 shows a graphical representation of the system model. A transition labelled with $\frac{\text{[cond] act}}{\text{upd}}$ means that the action act can be triggered if the condition cond is satisfied and triggering the action will result in the upd update. Update is optional and no update is denoted with a dash (—).

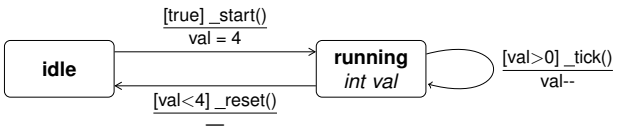

Figure 1. The countdown example: a high-level description.

A. Models

The approach used is based on models: the *system model* represents the behaviour of the system and the *mental model* is an abstraction of the system for the human operator. Those models are mathematically represented with enriched *labelled transition systems* (LTS) called HMI LTS, that are essentially graphs whose edges are labelled with actions. There are three kind of actions: *commands* are triggered by the user on the system, *observations* are actions autonomously triggered by the system but that the user can observe and finally *internal actions* are neither controlled nor observed by the user. Figure 2 shows the corresponding HMI LTS for the countdown example, where `_start()` and `_reset()` are considered as commands (solid lines) and `_tick()` is considered as an observation (dashed lines). Moreover, states of the models may be partitioned into modes in which

case each state is associated with a mode. Such additional information is useful to deal with mode confusion.

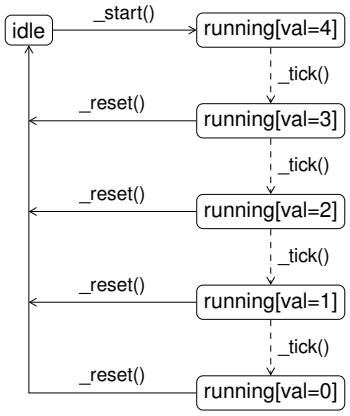

Figure 2. The corresponding HMI LTS of the countdown example.

B. Full-control property

The *full-control property* [5] ensures that at any point during system execution, the operator has sufficient knowledge about the system state to choose the proper control commands. Intuitively, it says that at any time during the interaction between the user and the system, the commands that are possible on the system must be known *exactly* by the operator and he must be aware of at least all the observations that the system may produce.

In other words, a mental model satisfying the property ensures that after any sequence of actions that can be executed on the system and on the mental models, the set of available commands on both models are the same, and the set of available observations according to the mental model is a superset of those available on the system model.

Figure 3 shows the minimal full-control mental model for the countdown example.

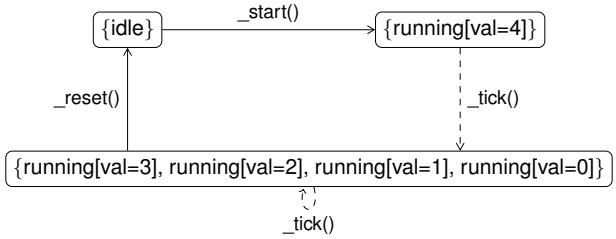

Figure 3. Minimal mental model of the countdown example (obtained with the reduction-based technique).

C. Mental model generation

Two algorithms, described in [5], [6], aim at generating automatically a minimal mental mental for a given system model. The first one is based on a bisimulation-based relation between the states of the system. Related states exhibit the same behaviour according to the standpoint of the operator and can therefore be merged together in the mental model. The second technique is based on a learning algorithm that iteratively builds mental model guesses, based on a teacher that is used to answer whether proposed execution sequences must, may or cannot be part of the mental model.

Whenever it is not possible to generate such a mental model for a given system, the algorithms provide diagnostic information in terms of a counterexample that illustrates problematic interactions which can be used by the designer to identify potential issues in the system model. Such situations occur when the system model is not *full-control deterministic*, meaning that the system can non-deterministically reach states that do not have the same set of possible commands. As a result, an operator would not be able to control such a system appropriately.

III. The jpf-hmi Extension

The `jpf-hmi` project uses the `jpf-statechart` framework [9], which represents hierarchical state machines as nested `State` class structures that can be executed and hence model checked by JPF.

It is important to understand the different models that are implicated at different levels. The first level is the one used in the statechart [10] model of the system (Figure 1 shows the two states of the countdown example). That statechart is translated into a Java program that is usable by the `jpf-statechart` extension. The `jpf-hmi` extension performs this translation for a specified statechart format. Each state of the statechart corresponds to a subclass of the `State` class provided with the `jpf-statechart` extension. That Java program is then executed by the JPF model-checker which retains the states of the execution, that is, states of the JVM. Finally, the `SC2LTS` tool that is presented hereafter computes the corresponding HMI LTS whose states correspond more or less to the JVM states (minor abstractions notwithstanding).

The `jpf-hmi` extension consists of a combination of different parts illustrated on Figure 4. The models to be analyzed can be provided in three different ways:

- An `XMI` file describing a statechart that can be exported from usual modelling tools like ArgoUML for example. The XMI file is parsed and is translated into a Java statechart program (`XMIParser`);
- A Java program following the `jpf-statechart` extension conventions;
- A `txt` file containing the explicit description of the HMI LTS. Such a file can be generated by the `jpf-hmi` extension (`LTSLoader`).

Some of the analysis algorithms that are provided in the extension require the fully expanded HMI LTS to work. The Java statechart is translated into an HMI LTS thanks to JPF and the `jpf-statechart` extension (`SC2LTS`).

Three types of analysis can be performed on HMI LTSs:

- The `FCCheck` algorithm checks whether a given mental model ensures full-control of a system model by an operator. If it is not the case, an execution trace is provided as a counterexample;
- The `Bisim` algorithm generates a minimal full-control mental model for a given system model, using the reduction-based technique. If such a mental model cannot be computed, then some diagnostic information is produced;
- The `Learning` algorithm does the same as the `Bisim` algorithm but uses the learning-based technique. In addition, when a mental model cannot be computed, it produces an execution trace that illustrates the problem as a counterexample.

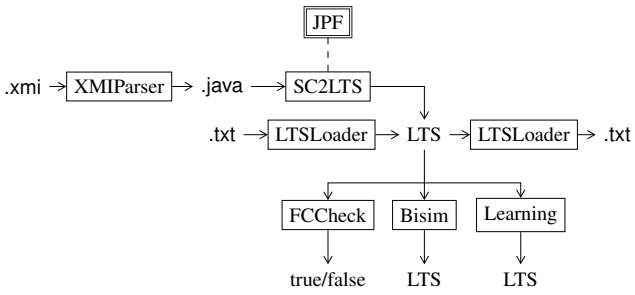

Figure 4. Overview of the `jpf-hmi` extension.

A. Describing the behaviour of models

The `jpf-hmi` extension includes an abstract `Model` class that represents an HMI LTS. Concrete input models are provided by means of extending this `Model` class, which require definition of the following class features:

- The `public List<Action> getActions()` method must be defined and returns the alphabet of the LTS with the partition into commands and observations;
- An inner class named `Behaviour`, which extends the `State` class, describes the behaviour of the model as a statechart according to the convention of the `jpf-statechart` framework.

The class may optionally contains a `public List<String> getModes()` method which returns the list of modes of the system. Figure 5 shows the Java program for the countdown example. There are three possible actions: the user can press the `_start()` button to activate the countdown (command), he can observe a `_tick()` whenever the countdown is running and the remaining time is decreased (observation) and he can reset the countdown with the `_reset()` button (command).

B. Performing analyses

To perform analyses within the `jpf-hmi` extension, the most convenient way is to use the `RunHMIAnalyzer` JPF shell which is configured with various options. The default

```
public class CountDown extends Model {
   @Override
   public List<Action> getActions() {
       List<Action> actions = new ArrayList<Action>();
       actions.addAll (Arrays.asList (
           new Action ("_start()", COMMAND),
           new Action ("_reset()", COMMAND),
           new Action ("_tick()", OBSERVATION)
       ));
       return actions;
   }

   public static class Behaviour extends State {
       private static final int MAX = 4;

       public class Idle extends State {
           public void _start() {
               running.val = MAX;
               setNextState (running);
           }
       } Idle idle = makeInitial (new Idle());

       public class Running extends State {
           int val = 0;

           public void _tick() {
               if (val > 0) {
                   val--;
                   setNextState (this);
               }
           }

           public void _reset() {
               if (val < MAX) {
                   setNextState (idle);
               }
           }
       } Running running = new Running();
   }
}
```

Figure 5. Java program of the countdown example's system model.

operation of the `RunHMIAnalyzer` tool is to generate the HMI LTS for the provided system model. Figure 6 shows the JPF application property configuration file that is used to generate the HMI LTS for the countdown example. The HMI LTS is computed and saved in two formats: a `.lts` file with the explicit description of the LTS and a `.dot` file with the graph of the system.

```
@using jpf-statechart

shell=gov.nasa.jpf.hmi.tools.RunHMIAnalyzer
+jpf-hmi.native_classpath=/tmp;

hmi.system=Countdown
hmi.system_output=/tmp/Countdown-system
```

Figure 6. JPF configuration file to perform analyses of the countdown example with the `jpf-hmi` extension.

Figure 2 shows the graph that is generated by the `SC2LTS` tool. The name of each generated state is the following: the name of the corresponding statechart state, together with all the instance variables with their corresponding values.

In order to generate minimal mental models, it suffices to define the `hmi.algo` option to `reduction` in order to use the reduction-based algorithm and to `learning` for using

the learning-based one. Figure 3 shows the mental model obtained with the reduction-based technique.

The configuration file to obtain this result is the same as the one of Figure 6 with two additional lines:

```
hmi.algo=reduction
hmi.mental_output=Countdown-mental
```

As discussed earlier, whenever it is not possible to compute a full-control mental model, the `jpf-hmi` extension provides a problematic interaction. For example, let us assume that we add one transition to the countdown model, with the internal unobservable action τ from state `running[val=0]` to `idle`, corresponding to an automatic reset. With this modification, the model is no longer full-control deterministic. As a result, it is not possible to get a full-control mental model and the `jpf-hmi` extension, when run with the learning algorithm, reports the following problematic interaction:

```
System model is not full-control deterministic
CEX: [_start(), _tick(), _tick(), _tick(), _tick()]
```

Indeed after executing the above sequence on the system, it is possible to reach two different states (`running[val=0]` and `idle`) which do not have the same set of enabled commands ($\{$`_reset()`$\}$ and $\{$`_start()`$\}$).

In order to analyse mode confusion issues, a mode must be associated to the states of the system. That association is done through the `@Mode` annotation on the classes representing states. If an issue occurs while computing a minimal mental model, and that issue is due to a mode, the reported error will clearly indicate a mode confusion issue. The way modes are handled in our framework is that loop transitions are added to all the states of the HMI LTS, labelled with commands corresponding to modes as discussed in [6].

C. Model abstraction

Realistic HMI models are often prohibitively large. Abstraction is therefore a crucial capability to improve both automatic analysis and readability for human operators. The `jpf-hmi` project currently supports three abstraction mechanisms that can be used to reduce the number of system states.

The first mechanism uses `FilterField` annotations for variable values that should be ignored when JPF matches program states, which constitutes a static filter. The second mechanism uses dedicated `Abstraction` objects to dynamically map ranges of concrete field values into abstract ones that are passed on the JPF matcher, which usually represents an under-approximation to reduce the number of explored system states. The third mechanism is to apply the abstraction a posteriori on the full HMI LTS, renaming states with the abstract values and merging them together. The latter results in an over-approximation of the system behavior. The drawback is that the full HMI LTS needs to first be created, which may not be possible for very large systems. On the other hand, it enables the simplification of system behavior for the human operator.

To implement dynamic runtime abstraction, the user has to provide concrete subclasses of JPF's `AbstractionAdapter` type, and mark the to-be-abstracted classes with `@Abstract("<qualified-abstraction-classname>")` annotations. Going back to the countdown example, one possible abstraction would be to not care about the exact value of the countdown's counter whenever it is running, but to only record whether the value is positive or equal to zero.

Figure 7 shows the class defining the abstraction. The `getAbstractValue` method takes as input the concrete value of the variable and returns the abstract value. The `getName` method takes a concrete value as input and returns a string representation for the abstract value. That latter method is optional and if it is not specified, the string representation will simply be the abstract value.

```
public static class ValAbs1 extends AbstractionAdapter {
   public int getAbstractValue (int v) {
       if (v > 0) {
           return 0;
       } else if (v == 0) {
           return 1;
       }
       return -1;
   }

   public String getName (int v) {
       int i = getAbstractValue (v);
       return i == 0 ? "(>0)" : "(=0)";
   }
}
```

Figure 7. Abstraction for the system model of the Countdown example.

To get an abstract version of the `Countdown` example, the first step is to annotate the `val` variable with `@Abstract("ValAbs1")`. The second step is to activate the abstraction mechanism through the configuration file by adding two lines to it:

```
vm.serializer.class=
          .jvm.serialize.DynamicAbstractionSerializer
das.classes.include=CountDown$Behaviour*
```

Figure 8 shows the LTS that is generated by the `SC2LTS` tool. The LTS is reduced to a two-state LTS which is an under-approximation of the concrete LTS (Figure 2). The `running[val=4]` state has been abstracted into `running[val=(>0)]`. The next state which is `running[val=3]` also corresponds to the `running[val=(>0)]` abstract state and the exploration by JPF is thus ended.

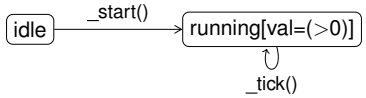

Figure 8. Under-abstraction of the countdown example, with the `jpf-core` abstraction built-in mechanism.

As discussed above, `jpf-hmi` also supports the generation of over-approximating system abstractions. For our example,

this could be achieved during the labelling of the states by the SC2LTS tool, through the getName method of the abstraction class. Figure 9 shows the abstraction class.

```
public static class ValAbs2 extends AbstractionAdapter {
   public int getAbstractValue (int v) {
       return v;
   }

   public String getName (int v) {
       return v > 0 ? "(>0)" : "(=0)";
   }
}
```

Figure 9. Second abstraction for the system model of the Countdown example.

Figure 10 shows the generated LTS. As already discussed, the disadvantage of the latter approach is that the concrete model will be completely explored by JPF, which is not the case with the other technique.

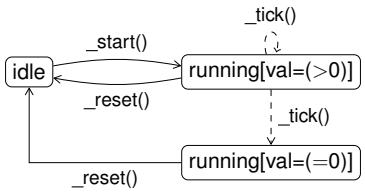

Figure 10. Over-abstraction of the countdown example, with the SC2LTS abstraction mechanism.

IV. Conclusion

This paper presented jpf-hmi, an extension of JPF that aimed at the analysis of systems where the focus is on interactions of human operators with the system. The modelling part is based on the jpf-statechart extension and the supported analysis capabilities implement algorithms discussed in previous work [5], [6], [8] on mental model generation.

The extension is currently a prototype for performing basic HMI system description and analysis. It is under active development, and several improvements are planned. In the future, the extension will use the JPF model-checking capabilities in order to add support for other standard HMI properties that can be expressed as model-checking problems (for example as safety properties). Moreover, we would like to extend and enhance the abstraction capabilities; for example, abstraction classes could be generated automatically in some cases. We may also try to provide some support for abstraction refinement. Finally, we plan to improve the XMI to JPF statechart translation in order to handle statechart models developed with tools like ArgoUML, as well as to connect the extension to HMI modelling tools such as ADEPT [11]. This will give us the opportunity to obtain significant case studies for our research in this domain.

References

[1] A. Degani, *Taming HAL: Designing Interfaces Beyond 2001*. Palgrave Macmillan, Jan. 2004.

[2] N. G. Leveson and C. S. Turner, "Investigation of the therac-25 accidents," *IEEE Computer*, vol. 26, no. 7, pp. 18–41, Jul. 1993.

[3] E. Palmer, "Oops, it didn't arm. — a case study of two automation surprises," in *Proceedings of the 8th International Symposium on Aviation Psychology*, 1996, pp. 227–232.

[4] M. Heymann and A. Degani, "Formal analysis and automatic generation of user interfaces: Approach, methodology, and an algorithm," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 49, no. 2, pp. 311–330, Apr. 2007.

[5] S. Combéfis and C. Pecheur, "A bisimulation-based approach to the analysis of human-computer interaction," in *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'09)*, G. Calvary, T. N. Graham, and P. Gray, Eds. New York, NY, USA: ACM, 2009, pp. 101–110.

[6] S. Combéfis, D. Giannakopoulou, C. Pecheur, and M. S. Feary, "Learning system abstractions for human operators," in *Proceedings of the 2011 International Workshop on Machine Learning Technologies in Software Engineering (MALETS 2011)*, Nov. 2011.

[7] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," in *Proceedings of the IEEE International Conference on Automated Software Engineering*, 2000, pp. 3–12.

[8] S. Combéfis, D. Giannakopoulou, C. Pecheur, and M. S. Feary, "A formal framework for design and analysis of human-machine interaction," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, Oct. 2011.

[9] P. C. Mehlitz, "Trust your model - verifying aerospace system models with JavaPathfinder," in *Aerospace Conference, 2008 IEEE*, Mar. 2008, pp. 1–11.

[10] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, Jun. 1987.

[11] M. S. Feary, "A toolset for supporting iterative human – automation interaction in design," NASA Ames Research Center, Tech. Rep. 20100012861, Mar. 2010.