Chapter 1
A First Introduction to Uppaal

Frits Vaandrager

Abstract This chapter provides a first introduction to the use of the model checking tool Uppaal. Uppaal is an integrated tool environment that allows users to model the behavior of systems in terms of states and transitions between states, and to simulate and analyze the resulting models. Uppaal can also handle real-time issues, that is, the timing of transitions. Using an example of a jobshop, we explain in a step by step manner how one can make a simple Uppaal model, simulate its behavior and analyze its properties. This introduction is targeted at a broad audience, ranging from high school students to software engineers and researchers. We only require elementary knowledge of programming and mathematics. Although a rich theory of model checking has been developed over the last decades, which includes both clever algorithms and deep mathematics, this introduction focuses entirely on the application of model checking technology, and hides the underlying mathematics from the reader.

1.1 Introduction

1.1.1 Model Checking

Model checking [6, 4, 1] is a powerful technique for automated debugging of complex reactive systems such as hardware components, embedded controllers and network protocols. In model checking, specifications about a system are expressed as (temporal) logic formulas, and efficient algorithms are used to traverse the model defined by the system and check if the specification holds or not. In 2007, E.M. Clarke, E.A. Emerson and J. Sifakis were awarded the ACM Turing Award for their

Frits Vaandrager
Institute for Computing and Information Sciences, Radboud University Nijmegen, Heijendaalseweg 135, 6525 AJ Nijmegen, The Netherlands, e-mail: `F.Vaandrager@cs.ru.nl`.

roles in developing model checking into a highly effective verification technology, widely adopted in industry [3].

Model checkers allow one to analyze models that capture the *dynamic behavior* of systems. These can be all sorts of systems: a network of computers or a printer, a Sudoku puzzle or an ant colony, an autonomous robot or train control software, etc. Actually, in principle any system can be analyzed using a model checker, as long as it has *states* and *transitions* between states. In order to illustrate the concept of model checkers, we consider the following puzzle:

Six girls each know a secret. By means of a series of bilateral conversations (regular phone conversations, say) they want to exchange all secrets. Whenever two girls have a conversation they share all the secrets they know at the time. How many conversations are needed before every girl knows every secret?

Figure 1.1 shows a *state-transition diagram* or *automaton* for the simplified version of the problem in which there are 3 girls. A *state* consists of a 3 by 3 matrix `knows` that records for each girl the gossips she knows. If girl i knows the gossip of girl j then we write a 1 in the entry for row i and column j: `knows[i][j] == 1`. Otherwise we write a 0: `knows[i][j] == 0`. In the *initial state*, at the top of the diagram, each girl only knows her own gossip, and hence there are 1's on the diagonal and 0's elsewhere. *Transitions* between states occur whenever girls have a conversation. In the initial state three transitions are possible: girls 1 and 2 call each other, girls 1 and 3 call each other, and girls 2 and 3 call each other. In the new states, the rows for the corresponding girls are adjusted and all gossips are exchanged. Not all conversations lead to a new state: sometimes a conversation does not produce any new information for any of the participants, and there is just a loop from the state to itself. Altogether 11 states can be reached starting from the initial state, and at least 3 conversations are needed before every girl knows every gossip.

Properties of state-transition diagrams can be described in the language of *temporal logic*. For instance, if φ is a property of states, then the temporal logic formula $\mathbf{E}\diamond\ \varphi$ describes the property "there exists a path that leads to a state in which φ holds". Model checkers compute whether a temporal logic formula holds for a given state-transition diagram. We can solve the gossiping girls puzzle by asking a model checker to produce the shortest diagnostic trace that shows that the formula $\mathbf{E}\diamond$ "each girl knows each gossip" holds for the state-transition model for 6 girls. In the syntax of the model checker Uppaal:

```
E<> forall (a : girls) forall (b : girls)
              knows[a][b] == 1
```

Figure 1.2 shows the solution for the gossiping girls puzzle that was found by Uppaal: at least 8 conversations are needed before every girl knows every secret. In fact, the message sequence diagram displayed in Figure 1.2 has been directly generated by the Uppaal tool.

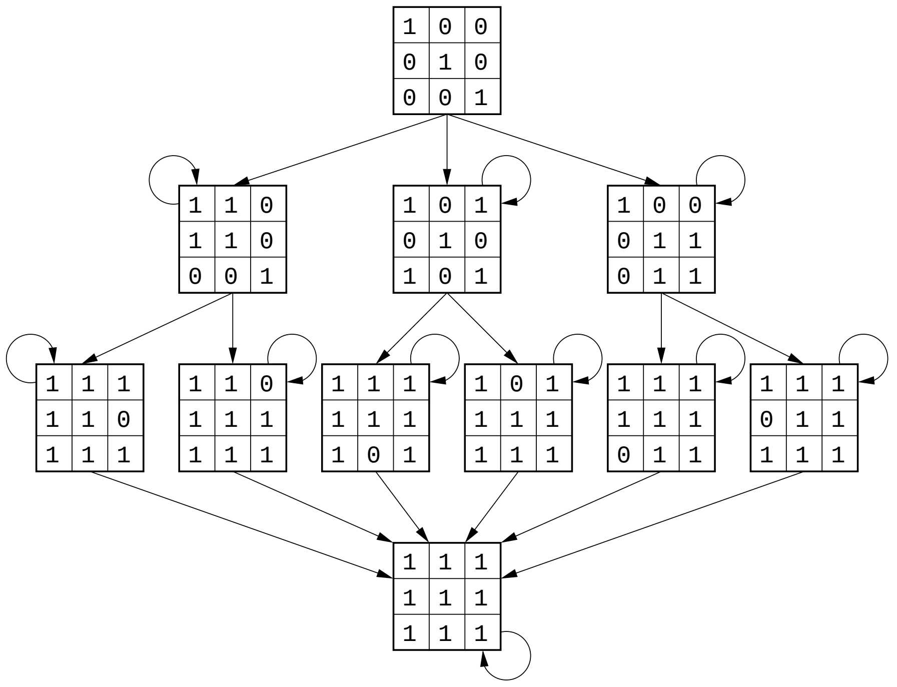


Fig. 1.1 State space for 3 gossiping girls.

Although utterly simple, the gossiping girls example already illustrates several key features of model checkers:

1. No proofs. Mathematicians have established that, in general, n girls need at least $2n-4$ conversations to exchange all gossips [10]. The result of [10] required ingenuity and hard work, whereas Uppaal produces its solution for the special case $n=6$ fully automatically.
2. Fast. Using his or her favourite model checker, an experienced user can construct a model for the gossip puzzle within half an hour. Uppaal needs a few minutes to find the optimal solution for $n=6$, and just a few seconds to find an optimal solution for the cases $n<6$.
3. Diagnostic counterexamples. Model checkers are very good at finding unexpected scenarios. When trying to solve the gossip puzzle, most people quickly come up with a solution that requires 9 conversations. The optimal solution found by Uppaal, which only requires 8 conversations, is tricky and much harder to find for humans. In industrial applications, model checkers often produce unexpected event orders that lead to an error state, fast solutions for scheduling problems, etc. In fact, since model checkers explore all reachable states, they will always produce a diagnostic counterexample (if it exists), provided they have been given

enough computational resources. The diagnostic counterexamples produced by model checkers frequently provide key insights in a design.

4. State space explosion. Since states of the model for n girls are $n \times n$ Boolean matrices, the collection of states that can be reached from the initial state via zero or more transitions (the "state space") will contain in the order of 2^{n^2} elements, and hence grows exponentially in n. Using some special verification features (symmetry reduction and the sweepline method), Uppaal can handle up to 7 girls (see Chapter 2 **Add ref**). Despite these limitations, in practice we often see that analysis of small instances of a parametrized model already produces important insights. It is trivial to generalize the solution with 8 conversations for $n = 6$ of Figure 1.2 to a general solution with $2n - 4$ conversations for $n > 6$. Similarly, if a design contains a flaw then usually this flaw can already be revealed by model checking a small instance of the design.

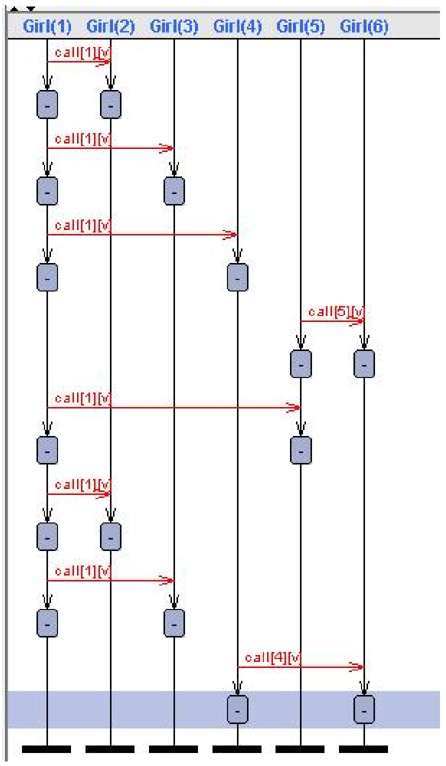


Fig. 1.2 Solution for gossiping girls puzzle found by model checker Uppaal.

1.1.2 Uppaal

Uppaal is an integrated tool environment that allows users to model the behavior of systems in terms of states and transitions between states, and to simulate and analyze the resulting models. Uppaal is available for free for non-commercial applications in academia and for private persons via `www.uppaal.org`. For commercial applications a commercial license is required, see `www.uppaal.com`. The software runs under Windows, Linux and Mac OS X. Uppaal is developed in collaboration between the Department of Information Technology at Uppsala University in Sweden and the Department of Computer Science at Aalborg University in Denmark, with input from several other universities around the world (including the author's group from the Radboud University Nijmegen).

Uppaal is a toolkit with a wealth of possibilities to model and analyze systems. In this chapter, we will restrict our attention to the absolute basics. The next chapter of this handbook will give an overview of some of the more advanced features of Uppaal. More details and documentation can also be found on the Uppaal website and in the tutorial paper [2] (the last article requires some background in formal methods). The help menu within Uppaal also provides an excellent explanation of the various features and possibilities of the tool. Actually, some text in this chapter has been directly taken from the help menu.

There are numerous other model checkers that one can freely download from the internet: Spin, Blast, MCRL2, Java Pathfinder, etc. We refer to [13] for an overview. Advantages of Uppaal are the graphical user interface and the short learning curve. After you have spent half a day on reading this tutorial and following the detailled instructions for building and analyzing some simple models, you can already start using the tool yourself.

1.2 A jobshop

We will now explain step by step how a simple production line can be modeled in Uppaal. This example is due to Robin Milner; we have taken the following description and illustration of Figure 1.3 from [12]:

We suppose that two people are sharing the use of two tools — a hammer and a mallet — to manufacture objects from simple components. Each *object* is made by driving a peg into a block. We call a pair consisting of a peg and a block a *job*; the jobs arrive sequentially on a conveyor belt, and completed objects depart on a conveyor belt. The jobshop could involve any number of people, whom we shall call *jobbers*, sharing more or fewer tools. In this chapter, we assume a system with two jobbers and a hammer and a mallet. To make the example more specific, we shall assume that the nature of the job influences the jobber's actions in a particular way. We suppose that he may use two predicates *easy* and *hard* over jobs, to determine whether a job is easy or hard or neither. He will do easy jobs with his hands, hard jobs with the hammer, and other jobs with either hammer or mallet.

Fig. 1.3 A jobshop (picture taken from [12]).

1.3 The system editor

After starting Uppaal, we see the window displayed in Figure 1.4. The Uppaal graphical user interface consists of three main parts, accessible via three tabs in the main window: the *system editor*, which can be used to construct models, the *simulator*, in which system behavior can be simulated, and the *verifier*, in which system behavior can be analyzed. In this subsection we discuss the system editor, subsection 1.4 will present the simulator, and subsection 1.6 will present the verifier. Upon starting Uppaal, at first the system editor is displayed.

A Uppaal model (called *system*) is defined as a composition of a number of basic components (called *automata* or *processes*). Automata are diagrams with states (called *locations*) and transitions between states (called *edges*).

The system editor has four drawing tools for building automata, see Figure 1.5, named *Select*, *Location*, *Edge* and *Nail*.

- The *Select tool* is used to select, move, modify and delete elements. Elements can be selected by clicking on them or by dragging a rubber band around one or

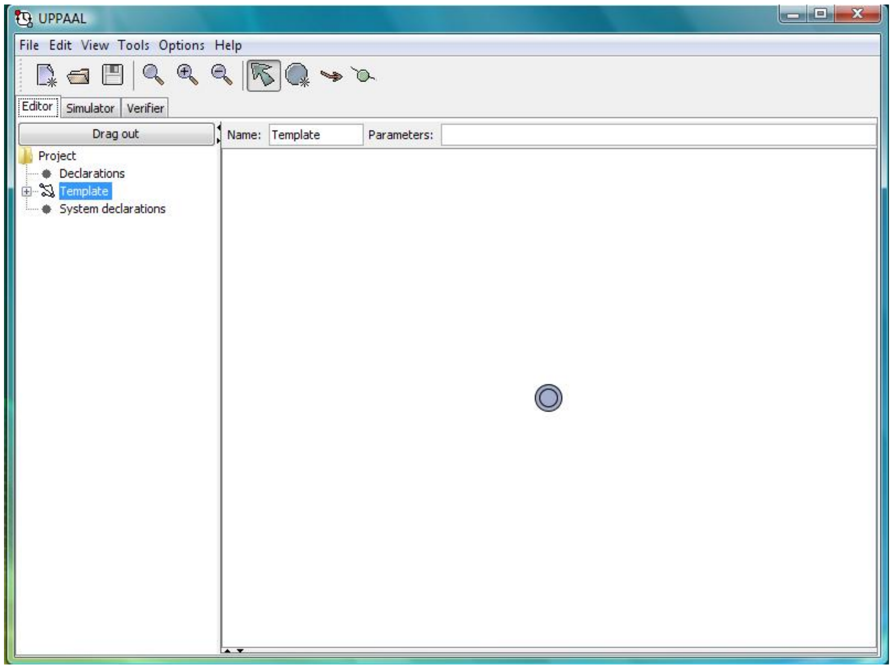


Fig. 1.4 Uppaal after starting the toolkit.

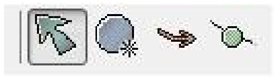


Fig. 1.5 The four drawing tools *Select*, *Location*, *Edge* and *Nail*.

more elements. Elements can be added or removed from a selection by holding down the control key while clicking on the element. The current selection can be moved by dragging them with the mouse. Double clicking an element brings up a menu in which properties for that element can be specified. Right clicking an element brings up a pop-up menu from which properties of the element can be changed.

- The *Location tool* is used to add new locations. Simply click with the left mouse button in order to add a new location. Be careful: if one clicks several times new locations will be stacked on top of each other (a common mistake, leading to models with strange behavior). In order to move a location to another position or to edit its properties, one first has to return to the *Select tool*.
- The *Edge tool* is used to add new edges between locations. Start the edge by clicking on the source location, then click in order to place nails and finally click the target location. The operation can be cancelled by pressing the middle or right mouse button. It is possible to change the source and target of an edge by

moving the mouse to the beginning or end of an edge until a small circle appears (the "nail"). Drag this circle to a new location in order to change the source or target of the edge.

- The *Nail tool* is used to add new nails to an edge, that is, places where an edge may change direction. Simply click and drag anywhere on an edge to add and place a new nail.

We will now construct our first Uppaal model. In the field *Name* at the top of the drawing window we enter the name of the first component ("template") of our model: `Jobber`. Next we right click (with the *Select tool*) on the location which Uppaal has already placed in the drawing window. We can then give this location a name, for instance `begin`. Each automaton has an *initial location*, marked by a double circle. During simulation or in verifications the automaton will always start in this location. By checking the box *Initial* in the menu for a location, we specify that this location is the initial location of the automaton. The menu contains a couple of other fields and options (*Invariant*, *Urgent* and *Committed*), but for the moment we will not use these. Note that in Uppaal there are no end states or final states.

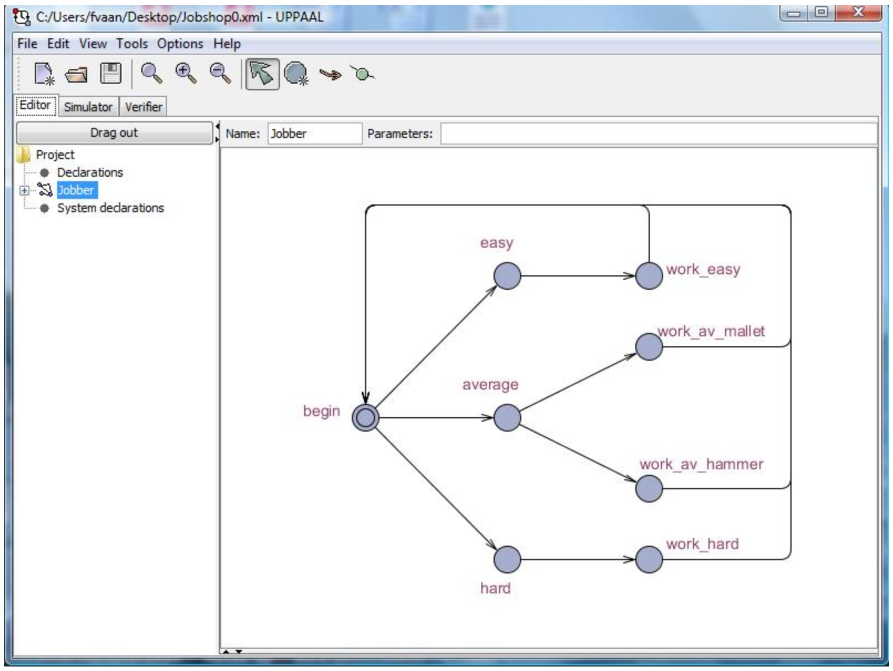
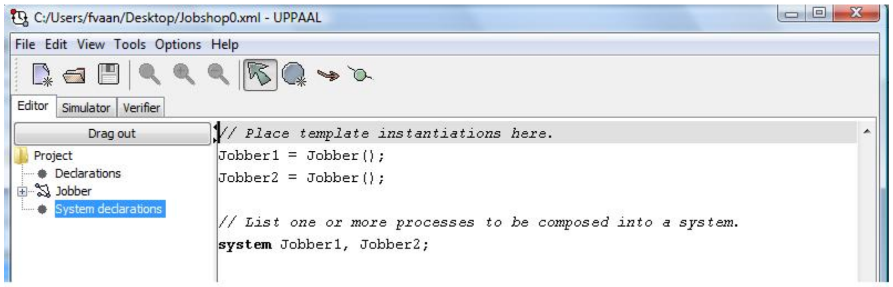


Fig. 1.6 A first version of the model.

We can continue drawing and construct the automaton that is depicted in Figure 1.6. The first transition from the initial location corresponds to the moment when a jobber picks a new job from the conveyor belt. There are three transitions possible since according to the informal specification there are three types of jobs: easy jobs, hard jobs and jobs with average difficulty (neither easy nor hard). The next transition corresponds to the moment at which the jobber grabs a tool (if needed) and starts working on the job. In the case of a job with average complexity, there are two possible transitions, depending on the tool that is selected. The third transition corresponds to the moment that the jobs is done: the automaton returns to its initial state and the jobber is ready for the next job.

At this point, we have not specified in our model how the choice between the transitions from location `begin` to location `easy`, `average` or `hard` is made. Also, we have not specified how the choice between the transitions from location `average` to location `work_av_mallet` and `work_av_hammer` is made. Choices for which the model does not specify how they are resolved are called *nondeterministic*. Nondeterminism is extremely useful for maintaining a high level of abstraction in descriptions of the behavior of physical systems and machines [8]. The nondeterministic nature of Uppaal is one of the most powerful features of the tool, it allows one to model behavior without having to specify conditions for all transitions, and to focus on the possible behavior in an abstract sense and still be able to check the model.


C:/Users/fvaan/Desktop/Jobshop0.xml - UPPAAL
File Edit View Tools Options Help
Editor
Simulator
Verifier
Drag out
Project
Declarations
Jobber
System declarations

```
// Place template instantiations here.
Jobber1 = Jobber();
Jobber2 = Jobber();

// List one or more processes to be composed into a system.
system Jobber1, Jobber2;
```

Fig. 1.7 Declaration of system components.

Once we have constructed the automaton of Figure 1.6, our first model is almost ready. Click on *System declarations* in the left window. We now see a screen in which one can list all the processes (automata) in a model. As shown in Figure 1.7, we specify that our first model consists of two instances of the template `Jobber`, called `Jobber1` and `Jobber2`. So our model consists of two automata that, intuitively, run in parallel. As we will see, a *global state* of a full model is fully determined by the locations (states) of its components, and the *global transitions* of a full model are in direct correspondence with the edges (transitions) of its components. By clicking on *Tools* in the menu bar at the top of the screen, and then on *Check Syntax*, we may check whether a model is syntactically correct. If a model contains mistakes, these will be underlined in red. A more detailed description, listing all the specific templates and declarations that contain syntax errors, is provided in a window at the bottom of the screen (normally not visible, but one can make it larger).

1.4 The simulator

Once a model is syntactically correct, we can simulate it, that is, explore the state space of the model in a step-by-step fashion, by selecting the tab *Simulator*. The resulting screen is displayed in Figure 1.8. Uppaal makes two copies of the template

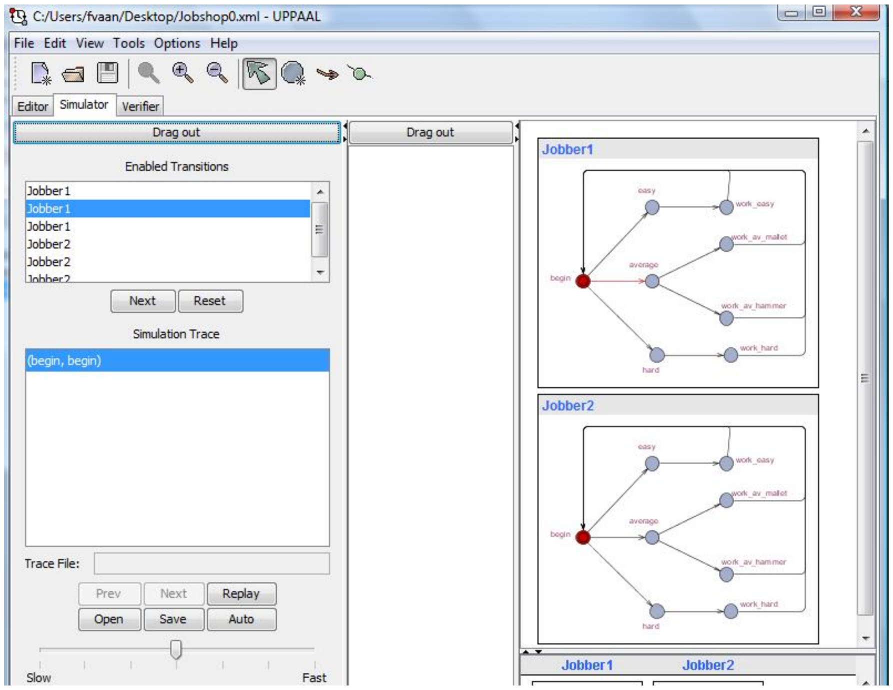


Fig. 1.8 Screenshot of the simulator.

`Jobber`, one for each automaton instance. Using red dots the current location of each automaton is highlighted. Initially, the current location of an automaton is its initial location. In the simulation control panel on the left, we see that (due to non-determinism) six transitions are possible from the initial state of the system (three for each jobber). When we select one of these transitions, the corresponding step in the automaton diagram for `Jobber1` or `Jobber2` is colored red. Pressing the *Next* button causes the simulated system to take the selected transition, and to update the current location. Using the *Prev* button one can go to the previous step in a simulation trace, with the *Reset* button one can bring the system back to its initial state, and with the *Replay* button one can instruct Uppaal to automatically replay the current trace. If we press the button *Auto*, then Uppaal starts executing randomly selected transitions, one after the other. The speed of the simulation can be changed by moving the arrow in between *Slow* and *Fast*. We can stop the random simulation by pressing the *Auto* button again.

Note that there exists a subtle difference between the nondeterministic choice of transitions as specified in Uppaal models and the probabilistic (random) choice of transitions by the simulator. Whenever a Uppaal model contains a nondeterministic choice, for instance the three outgoing transitions of location `begin`, we have no information whatsoever about how this choice will be resolved in an implementation: maybe, easy, average and hard jobs strictly alternate; maybe, a specific jobber always picks easy jobs; or maybe, arrival of jobs is governed by a probability distribution in which 50% of the jobs is easy, 40% is average, and 10% is hard. Etc. In general there are infinitely many ways to resolve a nondeterministic choice in a Uppaal model. Within the simulator, however, based on the idea that a user wants to see as many global states as possible during a simulation run, Uppaal uses a pseudorandom number generator to make a uniform probabilistic choice between all the transitions that are enabled within a given global state.

1.5 Channels

The simulator is very useful for obtaining insight in the behavior of a model and finding mistakes. By playing with our first model in the simulator, we quickly discover that something is wrong: both jobbers can be in location `work_hard` simultaneously. This should not be possible, since in this location both jobbers are using the hammer, and there is only one hammer. Hence we need to refine/correct our model.

In order to fix the model, we introduce separate templates for both the hammer and the mallet. For each tool there are 2 locations: `free` or `taken`. The automaton for a tool moves from location `free` to location `taken` when it is grabbed by one of the jobbers. In order to model the synchronization between tools and jobbers, we use the notion of *(synchronization) channels* from Uppaal. Once "`a`" has been declared as a channel, transitions can be labeled with either `a`! or `a`?. This can be done by double clicking (within the *Editor*) transitions with the *Select tool*, and then writing `a`! or `a`? in the *Sync* field. When two automata synchronize on channel "`a`", this means that an `a`! transition of one automaton occurs simultaneously with an `a`? transition of another automaton. An `a`! or `a`? transition can never occur on its own: `a`! always has to synchronize with `a`?, and vice versa. If there is one automaton `S` that can do an `a`! but two automata `R1` and `R2` that can do an `a`?, then there is a nondeterministic choice and `S` can synchronize with either `R1` or `R2`. The other automaton has do something else or has to wait until the next `a`! synchronization will be offered. In our jobshop example it does not matter which transition is labeled with a! and which transition is labeled with a?. Often, we place the a! on a transition of the component that takes the "initiative" for the synchronization. In the case of the jobshop, the jobbers take the initiative to grab a tool, whereas the tools are passive and just "wait" until someone is using them. Hence, we place the !'s in the template for the jobbers, and the ?'s in the templates for the tools. Figure 1.9 shows the adjusted model of the jobber, in which synchronization channels have been added. Synchronization channels must also be declared. This can be done by clicking on the

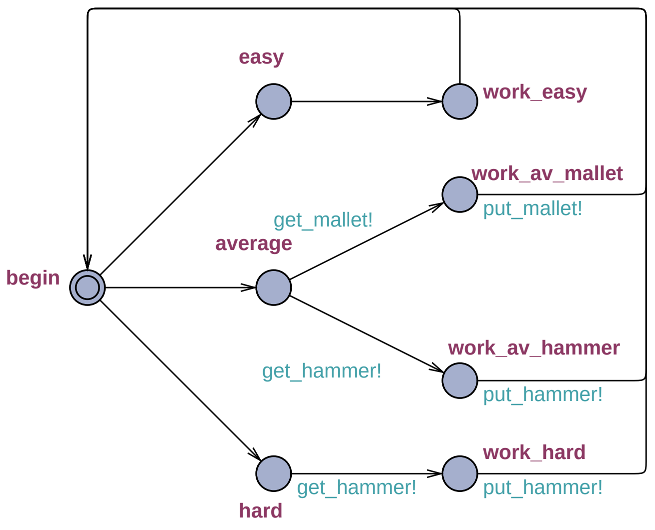


Fig. 1.9 Model of jobber, extended with synchronization labels.

(global) project *Declarations* in the window on the left, and inserting the following text:

```
// Place global declarations here.
chan get_hammer, put_hammer, get_mallet, put_mallet;
```

If we simulate the extended model, we quickly see that in the new model *deadlocks* are possible, global states from which no transition is possible. This occurs, for instance, when both jobbers are in location `hard` and both want to perform a `get_hammer!` transition. But since there is no automaton that can perform a matching `get_hammer?`, the system comes to a grinding halt. In order to rule out these deadlocks, we add new templates `Hammer` and `Mallet` (this can be done by selecting the option *Insert Template* in the *Edit* menu). Figure 1.10 shows the definitions of these templates.[1] We may add the new templates to the *System declarations*

[1] Actually, it would be better to have just one template `Tool`, with two instances `Hammer` and `Mallet`. It is possible to define this in Uppaal, but this involves adding channel names as param-

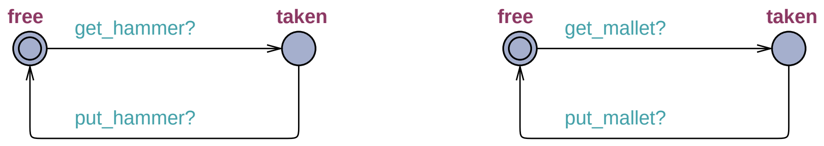


Fig. 1.10 Models of hammer and mallet.

via the text

```
system Jobber1, Jobber2, Hammer, Mallet;
```

Note that a system declaration may contain both names of instances of a template (`Jobber1` and `Jobber2`) as well as names of templates (`Hammer` and `Mallet`). We have now completed our first Uppaal model! We can open the model in the simulator and convince ourselves that it indeed behaves as specified in the informal description. For this it may be helpful to use the Message Sequence Chart (MSC) visualisation, which is available through the lower rightmost panel of the simulator. Figure 1.11 shows a screenshot with an MSC of our model. In the MSC view there is a vertical line for each automaton, and a horizontal line for each synchronization point. Once we are satisfied with the model, we can save it by selecting in the *File*

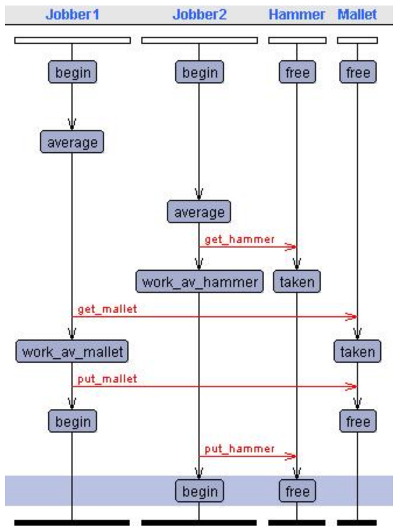


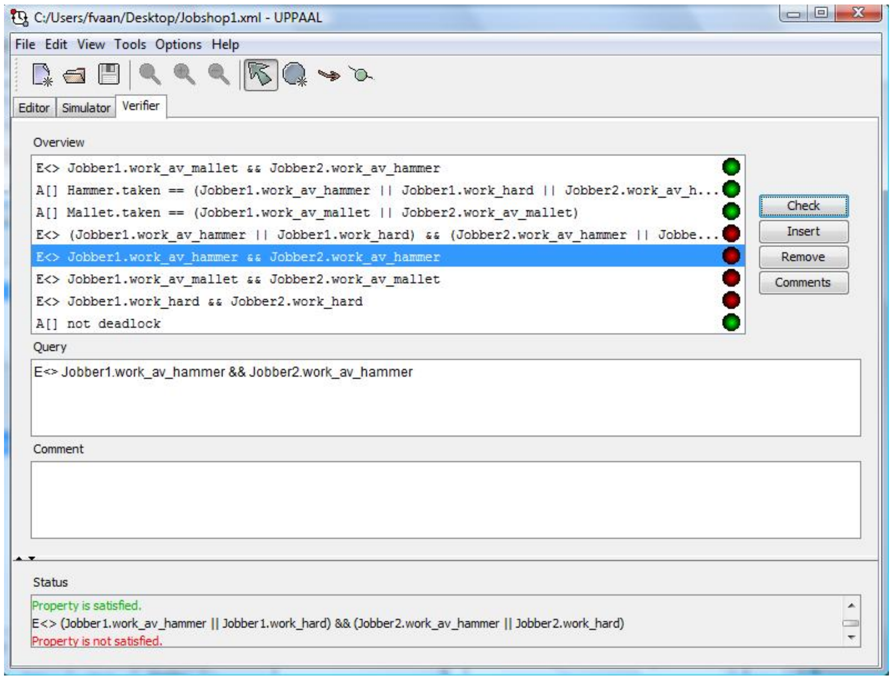
Fig. 1.11 Screenshot of the Message Sequence Chart panel.

eters to a template. Since we prefer to explain the notion of template parameters in Section 1.9 of this chapter, we introduce separate templates for hammer and mallet.

menu the option *Save System As...*. Uppaal models are stored as `.xml` files.

1.6 The verifier

The *Simulator* is very useful for playing with a model and obtaining insight, but it does not answer questions like “Is it possible to reach a state in which (some given) property *Bad* holds”? Even if we have not seen a *Bad* state after hours of simulation, this does not guarantee that no such state exists! Fortunately, Uppaal’s *Verifier* allows us to rigorously answer this type of questions.


C:/Users/fvaan/Desktop/Jobshop1.xml - UPPAAL
File Edit View Tools Options Help
Editor
Simulator
Verifier
Overview
Check
Insert
Remove
Comments
Query
Comment
Status

```
E<> Jobber1.work_av_mallet && Jobber2.work_av_hammer
A[] Hammer.taken == (Jobber1.work_av_hammer || Jobber1.work_hard || Jobber2.work_av_h...
A[] Mallet.taken == (Jobber1.work_av_mallet || Jobber2.work_av_mallet)
E<> (Jobber1.work_av_hammer || Jobber1.work_hard) && (Jobber2.work_av_hammer || Jobbe...
E<> Jobber1.work_av_hammer && Jobber2.work_av_hammer
E<> Jobber1.work_av_mallet && Jobber2.work_av_mallet
E<> Jobber1.work_hard && Jobber2.work_hard
A[] not deadlock
```

```
E<> Jobber1.work_av_hammer && Jobber2.work_av_hammer
```

```
Property is satisfied.
E<> (Jobber1.work_av_hammer || Jobber1.work_hard) && (Jobber2.work_av_hammer || Jobber2.work_hard)
Property is not satisfied.
```

Fig. 1.12 Screenshot of the *Verifier*.

1.6.1 Queries

Within the *Verifier*, we can specify a so-called *Query*, a property that may or may not hold for a given model. By exhaustive exploration of the set of reachable states (the state space), the *Verifier* can establish whether a *Query* is satisfied or not. Figure 1.12

shows a screenshot of the *Verifier*. Queries often start with the symbols "`A[]`". This notation, which is taken from the field of temporal logic, means "In all reachable states it is the case that". Thus, for example, the query

```
A[] not deadlock
```

states that in all reachable states of the system there is no "deadlock". Recall that a global state has a "deadlock" if no outgoing transitions are possible. Stated differently, the above query asserts that in all reachable states at least one transition is possible. If we type the query in the *Query* window of the *Verifier* and then press the *Check* button, Uppaal explores all the reachable states to see if they have an outgoing transition. In the case of our model this is indeed the case, and therefore Uppaal returns *Property is satisfied*. This means that in each of the reachable states always either `Jobber1` or `Jobber2` can proceed.

A new query can be entered by pressing the *Insert* button, and typing the query in the window *Query*. We may for instance enter the following text:

```
E<> (Jobber1.work_hard && Jobber2.work_hard)
```

Here the notation "`E<>`", again taken from temporal logic, means "There exists a reachable state such that". The above query asserts that there exists a reachable state in which `Jobber1` and `Jobber2` are in location `work_hard`, that is, both jobbers are working on a hard job. Much of the syntax of Uppaal is similar to that of common programming languages such as C, C++, Perl and Java. When we ask Uppaal to check the above query, the result is *Property not satisfied*. This is what we expect: if a jobber is working on a hard job he is using the hammer, and since there is only one hammer, at most one jobber can work on a hard job at a time. Note that Uppaal does not use any clever form of reasoning to arrive at this conclusion. The tool just uses brute force to explore all the reachable global states of the model and to check for each of these states whether both jobbers are working on a hard job.

In the Uppaal help menu the full syntax for queries and expressions is described. In this chapter, we only consider queries of the form `A[]e` and `E<>e`, where `e` is an expression. An expression `e` consists of a boolean combination of atomic propositions. Atomic propositions can be of the form `A.l`, for `A` an automaton and `l` a location. Such a proposition is true in a global state of the model if in this state automaton `A` is in location `l`. Table 1.1 gives some examples of boolean operators that can be used in Uppaal to define properties. Further on in this chapter we will encounter other operators for constructing properties.

Symbol	**Operator name**	**Meaning**
`&&`	and	`e && f` is true if both `e` and `f` evaluate to true
`||`	or	`e || f` is true if `e` evaluates to true or `f` evaluates to true
`==`	equality	`e == f` is true if `e` and `f` evaluate to same value
`imply`	implication	`e imply f` is true if `e` evaluates to false or `f` evaluates to true
`not`	negation	`not e` is true if `e` evaluates to false

Table 1.1 Some logical operators in Uppaal

1.6.2 Diagnostic traces

We can ask Uppaal whether there exists a reachable state in which one jobber is working on an average job with a mallet, and the other jobber is working on a average job with the hammer:

```
E<> (Jobber1.work_av_mallet && Jobber2.work_av_hammer)
```

Uppaal then answers *Property is satisfied*. In this case, Uppaal can also provide a concrete example that illustrates why the property holds, that is, a trace leading to a state in which the first jobber is working on an average job using the mallet, and the second jobber is working on an average job using the hammer. In order to let Uppaal compute such an example, we choose under *Options* the entry *Diagnostic Trace* and then select the option *Shortest*. If we now let Uppaal check the above property again, it will again produce the answer *Property is satisfied* but in addition it will compute the shortest path (or trace) leading to a state in which both `Jobber1.work_av_mallet` and `Jobber2.work_av_hammer` hold. The tool asks whether it may store this path in the simulator. If we give Uppaal permission to do this, we can replay the trace in the simulator. If we open the simulator, then we see the final state of the trace in which `Jobber1` is in location `work_av_mallet` and `Jobber2` is in location `work_av_hammer`. By pressing the *Reset* button we go to the initial state of the trace, and by pressing the *Replay* button we tell Uppaal to replay the trace within the simulator. We can also replay the trace step-by-step by repeatedly pressing the *Next* button at the bottom of the screen.

In general, Uppaal can provide a diagnostic trace for `E<>` properties that hold, and for `A[]` properties that do not hold. In the case of `E<>` properties that do not hold, or `A[]` properties that hold, Uppaal can only report that it exhaustively checked all the reachable states of the model and didn't find anything. In the above example of an `E<>` property, the diagnostic trace is rather trivial and consists of only four transitions. However, in realistic models of industrial applications, diagnostic traces may describe tricky scenarios involving thousands of transitions. In such cases, Uppaal's ability to provide diagnostic traces is essential (see [7] for an example where Uppaal produced a diagnostic trace for a protocol of Bang & Olufsen involving more than 2000 steps). If Uppaal says that a certain correctness property does not hold, an engineer typically wants to know why this is the case.

1.6.3 Saving queries and traces

We can save queries by selecting in the *File* menu the option *Save Queries As...*. The queries are then saved as a text file with extension `.q`. If one opens a Uppaal model `model.xml`, then automatically also the query file `model.q` is opened (if it exists). Alternatively, one can open a query file by selecting in the *File* menu the option *Open Queries...*. It is also possible to save traces by pressing the *Save* button in the simulator. Traces are saved as (unreadable) text files with extension `.xtr`. A trace file can be opened by pressing the *Open* button in the simulator.

1.6.4 How many states are there?

Let us try to compute the total number of reachable states of our jobshop model. Since each jobber is modeled by an automaton with 8 locations, and each tool is modeled by an automaton with 2 locations, there are at most $8 \times 8 \times 2 \times 2 = 256$ global states: the number of global states is bounded by the product of the number of states of all the components in the system. However, many of these global states can not be reached from the initial state: they will never occur in any run of the system. In order to see this, observe that once we know the state of the two jobbers, we also know the state of the two tools:

```
A[] Mallet.taken == (Jobber1.work_av_mallet
                    || Jobber2.work_av_mallet)

A[] Hammer.taken == (Jobber1.work_av_hammer
                    || Jobber1.work_hard
                    || Jobber2.work_av_hammer
                    || Jobber2.work_hard)
```

The first query states that the mallet is taken exactly when either the first or the second jobber uses it for an average job. The second query states that the hammer is taken exactly when either the first or the second jobber is using it for either an average or a hard job. Since the locations of both mallet and hammer are fully determined by the locations of the jobbers, this means that our model has at most $8 \times 8 = 64$ global states that are reachable from the initial state.

Five of these remaining 64 states can not be reached since the mallet and hammer can only be used by one jobber at a time:

```
A[]
not (Jobber1.work_av_mallet && Jobber2.work_av_mallet)
 &&
not (Jobber1.work_av_hammer && Jobber2.work_av_hammer)
 &&
not (Jobber1.work_av_hammer && Jobber2.work_hard)
 &&
not (Jobber1.work_hard      && Jobber2.work_av_hammer)
 &&
not (Jobber1.work_hard      && Jobber2.work_hard)
```

All the other combinations of locations of `Jobber1` and `Jobber2` can be reached, and so the total number of reachable states of our model is $64 - 5 = 59$. Unfortunately, the regular version of Uppaal has no option to count the number of reachable states in a model. Such an option is present however in the command line version of the verifier: `verifyta -u`. For Uppaal our model is really small: the tool can easily handle models with thousands or even millions of states. Nevertheless, it is also easy to construct models that are so big that Uppaal cannot handle them and, for instance, runs out of memory. For instance, if we modify our jobshop model and increase the number of jobbers to 100, then the size of the state space becomes in the order of 8^{100} and too big for Uppaal. When constructing Uppaal models, one should always take care that the state space does not become too big.

1.7 Variables

We will now describe how one can add integer *state variables* to Uppaal models. The values of these variables can be tested and updated in transitions, and thus influence the behavior. State variables are indispensable for modeling nontrivial systems and give the Uppaal modeling language an expressive power that is comparable to simple programming languages. To illustrate the use of state variables, we discuss a small modification of the jobshop model:

We suppose that the jobbers stop with their work as soon as they have completed 10 jobs together.

In order to capture this additional requirement in our model, we add an integer state variable that records how many jobs have been taken from the belt. This is achieved by adding the following lines in the global project *Declarations* in the window on the left in the *Editor*:

```
const int J = 10;
int[0,J] jobs;
```

The first line declares an integer constant with value 10. As suggested by their name, the value of constants always remains the same. In the second line an integer variable `jobs` is declared with minimum value 0 and maximum value J. The value of variables can be modified when transitions occur. By default, the initial value of a variable is 0. The domain of integer variables in Uppaal is always bounded. If we specify no bounds and simply declare

```
int jobs;
```

then implicitly the minimum value is -32768 and the maximum value is 32767.[2] Whenever during exploration of the state space — either in the simulator or in the verifier — a variable gets assigned a value outside its domain, this is referred to as a "run time error" and an error message is generated.

[2] Uppaal uses 16-bit `int`'s, including 1 bit to represent the sign.

Figure 1.13 shows an extension of the jobber model in which the variable `jobs` is used. In the location `begin`, a jobber only accepts a new job when `jobs` < `J`. In this case, the value of `jobs` is incremented by 1. We can implement this change of the model by double clicking the transition from `begin` to `easy`, and then write `jobs` < `J` in the field *Guard* of the resulting menu. In Uppaal, a transition can only be taken if its guard evaluates to true (if no guard is specified then we assume it equals `true`). In the field *Update* of the transition, we specify that the value of `jobs` must be incremented by 1 whenever the transition is taken. This is done by writing `jobs`++, using syntax that has been taken from the programming language C. Alternatively, we can also write `jobs` = `jobs` + `1` or `jobs` := `jobs` + `1` (this all means the same). We also add an extra location `done` to the model, and a transition from `begin` to `done`, which is taken whenever `jobs` has reached the value `J` and all jobs are done.

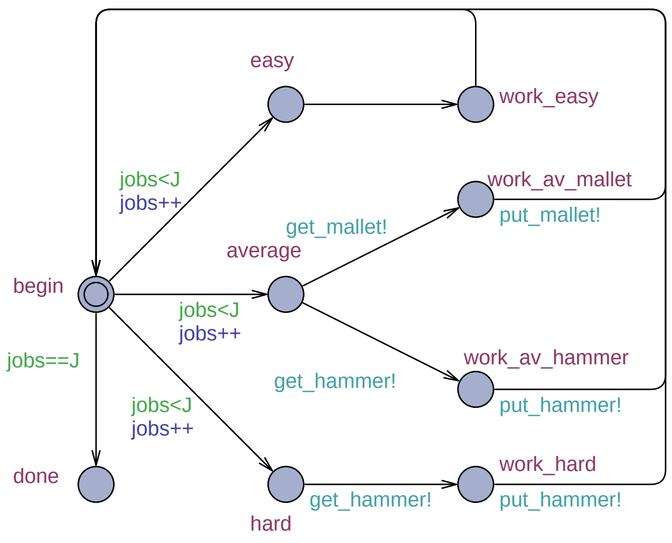


Fig. 1.13 Model of jobber in which exactly `J` jobs are carried out.

When we now start the simulator, we see a separate window in which, for each state, the value of the state variable `jobs` is listed. Using the verifier, we can establish that the model of Figure 1.13 satisfies exactly the same correctness properties as the model of Figure 1.9, except for the property

```
A[] not deadlock
```

which is no longer satisfied. When the jobbers have finished `J` jobs and have moved to location `done`, no further transitions are possible and therefore a deadlock state has been reached. We recall that in Uppaal deadlock is a property of global states, not of individual components. Thus a global state in which, for instance, `Jobber1` is in location `done` but `Jobber2` is in location `easy` is not deadlocked, even though for `Jobber1` no further transitions are possible.

Observe that in the modified model, both jobbers *together* perform `J` jobs. This is because `jobs` is a *global* variable that can be tested and updated by both jobbers. In Uppaal, we can also declare *local* variables, which can only be used by one automaton. When in the *Editor* we click on the "+" symbol at the left of the template `Jobber`, a new line *Declarations* appears below `Jobber`. When we select this line, we can declare local variables for the template. For instance, by moving the line

```
int[0,J] jobs;
```

from the global declarations section to the local declarations section of template `Jobber`, we give `Jobber1` and `Jobber2` each their own, local copy of variable `jobs`. The result is that both `Jobber1` and `Jobber2` have to perform `J` jobs, instead of `J` jobs together.

Uppaal has a rather extensive syntax for the expressions in guards and updates. It is possible to declare arrays and Boolean variables, and a user can even define record types and new functions. The syntax for doing this is very similar to the syntax of programming languages such as C and Java. For an overview of the syntax we refer to Chapter 2 of this handbook **Add ref** and to the help menu of Uppaal: click on *Language Reference* and then on *Expressions*.

1.8 Time and clocks

In the design and analysis of real systems, timing aspects often play a role. Sometimes we may abstract from quantitative timing, and only consider the ordering of events, but often timing information has to be included in the model in order to be able to answer certain questions. For instance, we may need to establish not only that a certain location can be reached, but also how fast. In the case of our jobshop, the following question could arise. Uppaal has been more or less designed to answer this type of questions.

We suppose that a jobber needs (at least) 5 seconds for an easy job, 10 seconds for an average job using the hammer, 15 seconds for an average job using the mallet, and 20 seconds for a hard job. We suppose that the jobs arrive in the following order: `H`, `A`, `H`, `H`, `H`, `E`, `E`, `A`, `A`, `A`, where `E` denotes an easy job, `A` an average job, and `H` a hard job. How much time do the two jobbers need (at least) to complete the 10 jobs?

1.8.1 Modeling the conveyor belt

Before we add timing to our model, we first add an automaton that describes the behavior of the conveyor belt. The belt was not included in our first model (there was no reason for adding it) but in order to answer the question about timing, the order in which jobs arrive appears to be relevant. In order to model incoming jobs, we declare three new channels `jobE`, `jobA` en `jobH`, which correspond to the arrival of easy, average, and hard jobs, respectively. The automaton `Belt`, which is depicted in Figure 1.14, describes the behavior of the conveyor belt that delivers the 10 jobs in the specified order. By adding automaton `Belt` to the model in *System declarations*

```
system Belt, Jobber1, Jobber2, Hammer, Mallet;
```

and by labeling the outgoing transitions of location `begin` of the jobber automaton of Figure 1.9 in the obvious way with the synchronization channels `jobE`?, `jobA`? en `jobH`?, we model that the jobbers have to deal with the specified sequence of 10 jobs.

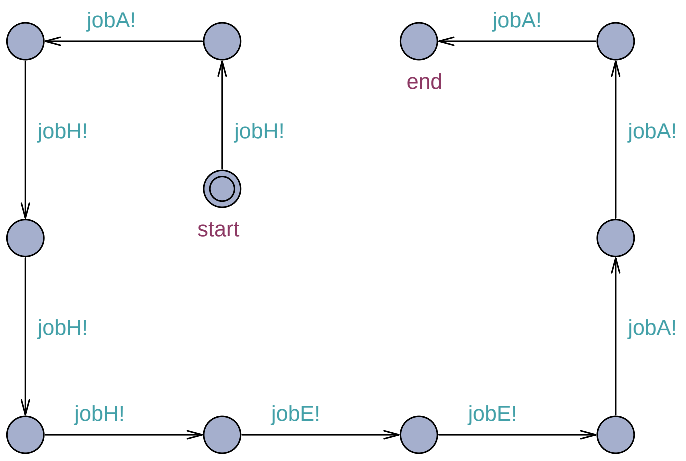


Fig. 1.14 Model of conveyor belt that delivers 10 jobs.

1.8.2 Clocks and lower bounds on timing

In the models that we have constructed thus far, time is not modeled explicitly. In Uppaal we assume that transitions occur instantaneously and do not take time. Time may only elapse when all automata in the model are waiting in a location. Whenever we want to model an activity that takes time, we can do this by introducing two consecutive transitions, one corresponding to the start of the activity, and another corresponding to the end of it. Often, models only contain transitions that correspond to either the beginning or the end of an activity that has a duration. If we want to be really precise then, for instance, we could say that the `jobH!` transition corresponds to the moment when a jobber starts to pick a hard job from the conveyor belt, the `get_mallet` transition corresponds to the moment when a jobber starts grabbing the mallet and the `put_mallet` transition corresponds to the moment when a jobber ends the activity of putting the mallet back on the table again.

If we want to specify lower and upper bounds on the time that an automaton may stay in a certain location, we can do this in Uppaal using so-called *clocks*. A clock is a special type of variable, whose domain consists of the set of nonnegative real numbers. Just like other variables, clocks can be declared either as a global variable (which can be tested and updated by all automata) or as a local variable (which can only be used by one automaton). In the initial state, all clocks have value 0. When an automaton is waiting in a location and time elapses then the values of its clocks increase. More precisely, when t time units pass then the values of all clocks in the model increase with t. Thus, all clocks are “perfect” and increase at exactly the same rate as time. In reality, of course, no clock is 100% perfect, but in our modeling language it is convenient to use the idealization of perfect clocks to specify upper and lower bounds on the timing of transitions.

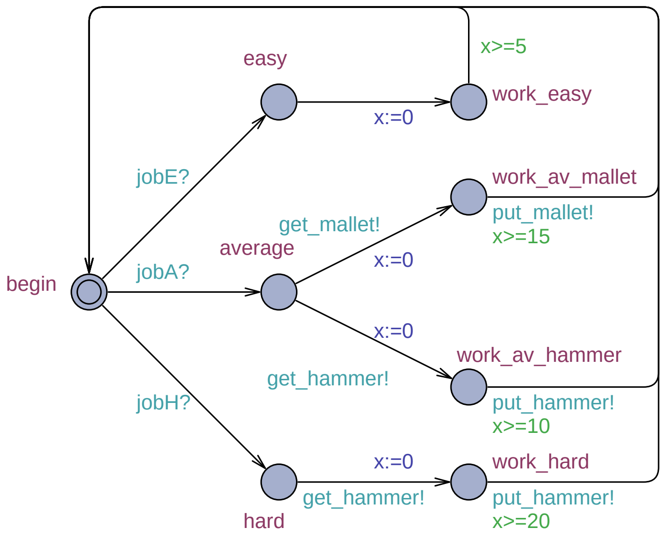


Fig. 1.15 Model of jobber extended with a clock.

Figure 1.15 shows an extension of the jobber model of Figure 1.9 with new synchronization channels `jobE`?, `jobA`? and `jobH`?, and also with a clock variable `x`.[3] Clock `x` has been declared by adding the following line to the local *Declarations* section of template `Jobber`:

```
clock x;
```

When a jobber moves from location `easy` to location `work_easy`, that is, starts to work on an easy job, clock `x` is reset to 0 via an update `x` := `0`. Subsequently, the guard of the outgoing transition of `work_easy` tests whether `x` >= `5`. In this way, we enforce that this transition may only occur once the automaton has been in location `work_easy` for at least 5 time units. This corresponds to our assumption that a jobber needs at least 5 time units to complete a simple job. In a similar way we model that the automaton spends at least 10, 15 and 20 time units in locations `work_av_hammer`, `work_av_mallet` and `work_hard`, respectively.

In the resulting model, `Jobber1` and `Jobber2` both have a local clock variable `x` that records how long they have been working on a certain job. Each time when a jobber starts with a new job its local clock is reset to 0. In order to record the total amount of time that has elapsed, we also introduce a global clock `now`, which is never reset. As a result, the global declarations look as follows:

```
chan jobE, jobA, jobH,
     get_mallet, get_hammer, put_mallet, put_hammer;
clock now;
```

We can ask Uppaal whether there exists a state in which all the 10 jobs have been delivered and both jobbers have returned to their initial state (that is, all the jobs have been completed):

```
E<> (Belt.end && Jobber1.begin && Jobber2.begin)
```

When Uppaal computes a diagnostic trace which demonstrates that this property is satisfied, this will, in general, not be the shortest trace. After all, we have only specified lower bounds for the timing in our model and no upper bounds. Thus the belt may wait indefinitely before delivering a job, a jobber may dawdle for days before he starts with the job, and for years before completing it. However, if we select in the menu *Options* under *Diagnostic Trace* the option *Fastest*, then Uppaal will produce the fastest execution that leads to the specified state.[4] In the simulator we can see that in the final state of this execution `now` >= `100`. This means that the jobbers need at least 100 time units to complete the 10 jobs. The easiest way to understand the schedule that has been computed by Uppaal is via the so-called "message sequence chart" in the lower right window of the simulator: here we see which jobber handles which job. Figure 1.16 visualizes this fastest schedule in an

[3] Note that this extension is orthogonal to the extension with the integer variable `jobs` displayed in Figure 1.13.

[4] If there are multiple executions with the same minimal duration, Uppaal will show the first one that it finds, which is always the same.

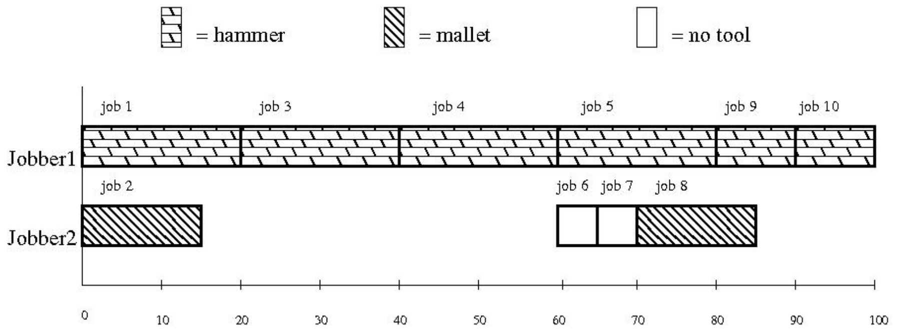


Fig. 1.16 Fastest schedule for completing the 10 jobs.

even more compact way as a "Gantt chart".[5] We see that one jobber is permanently busy with the hammer, whereas the other jobber has a relaxed schedule in which he is idling most of the time but also completes some jobs with his hands or with the mallet. It is easy to come up with an equally fast schedule in which the work load is more evenly distributed. We may decide, for instance, to give the third job to `Jobber2`.

1.8.3 Upper bounds on timing

We have seen that *lower bounds* on timing can be specified with the help of clock constraints in guards. For example, the constraint `x` $>=$ `5` in the guard of the transition from `work_easy` to `begin` indicates that this transition can only be taken once the jobber has spent at least 5 time units in location `work_easy`. In practice, we often need to prove upper bounds on timing: an airbag needs to inflate within a few microseconds after a collision has been detected, a soccer playing robot must quickly react when the ball is nearby, etc. In our jobshop example, the following question may arise:

We suppose that a jobber needs at most 7 seconds for an easy job, 12 seconds for an average job with the hammer, 17 seconds for an average job with the mallet, and 22 seconds for a hard job.

How much time do the two jobbers need at most to complete the 10 jobs, assuming that a jobber picks a new job from the belt as soon as he is ready to

[5] Figure 1.16 was constructed manually. Effort is underway to extend Uppaal with a feature for automatic visualization of traces using Gantt charts. This feature is already present in Uppaal Tiga, an extension of Uppaal for solving timed games, see `http://people.cs.aau.dk/~adavid/tiga/`.

work on it, and that he grabs a tool that allows him to do a job as soon as it becomes available?

In Uppaal we can specify *upper bounds* on timing, using so-called “invariants”. When we double click the location `work_easy`, a window appears with a field *Invariant*. By entering `x` $<=$ 7 in this field, we specify that in this location the value of `x` will always be at most 7. In other words, a jobber needs at most 7 time units to complete a simple job. If more than 7 time units have elapsed then the jobber has left location `work_easy`. In Figure 1.17 upper bounds of 7, 12, 17 and 22 have been added for locations `work_easy`, `work_av_hammer`, `work_av_mallet` and `work_hard`, respectively. In addition, an upper bound of 0 has been added for loca-

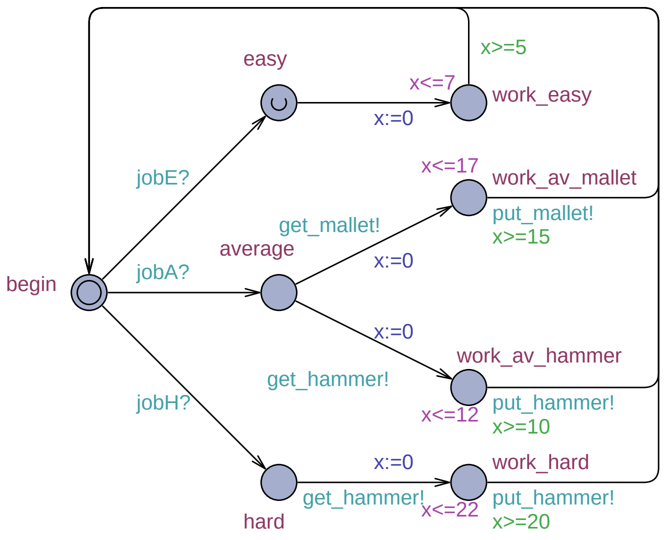


Fig. 1.17 Model of jobber with upper bounds on timing.

tion `easy`. This models the assumption that when a jobber has picked an easy job from the belt he starts working on it right away. We could have enforced this upper bound by resetting clock `x` upon entering location `easy`, in combination with an invariant `x` $<=$ 0 for this location. But Uppaal supports a simpler way of specifying the same requirement: if we double click the location `easy` we can check the field *Urgent*. If a location is *Urgent* then this means that time can not elapse within this location, and hence a transition to another location will occur immediately.

After all these modifications of the model, it is still not possible to infer an upper bound on the time need to complete all jobs. The reason is that workers may wait indefinitely before picking the next job from the belt, and they may wait indefinitely before grabbing a tool. We have not yet modelled the requirement that jobbers grab jobs and tools as soon as they can. A convenient way to eliminate idling is by making the synchronizations for grabbing a job or tool "urgent". When a synchronization channel is urgent, this means that whenever a synchronization with this channel is enabled, time can not advance and a transition has to be taken immediately. We can specify this in Uppaal by changing the declarations of the channels as follows:

```
urgent chan jobE, jobA, jobH, get_mallet, get_hammer;
chan   put_mallet, put_hammer;
```

Note that there exists a subtle difference between the use of urgent locations and urgent synchronizations. If we would make location `begin` of the jobber template urgent rather than channels `jobE`, `jobA` and `jobH`, a problem would arise in a state where a jobber is in location `begin` but no further jobs are on the belt and hence no `jobE`, `jobA` or `jobH` synchronizations are offered: in such a state there would be no possibility for time to progress and there would be a "time deadlock". Clearly, such a model is not realistic. Likewise, a time deadlock would arise if we would make location `hard` urgent rather than channel `get_hammer`.

Observe that in the model of Figure 1.17, a jobber may spend in between 5 and 7 time units in location `work_easy`. We have not specified in our model any further information about the time needed to complete an easy job: this choice is fully nondeterministic. Maybe a jobber usually completes an easy job within 6 time units but occasionally needs more time when he is tired. Maybe one jobber always completes a job within 5.731 time units, whereas another jobber always needs at least 6.194 time units. Maybe jobbers will always complete easy jobs within 5.5 and 5.8 time units, but we have not carried out the exact measurements and want to be on the safe side. The ability to have nondeterminism in the timing of transitions is an extremely useful feature of timed automata, which makes it possible to describe systems and reason about them at a high level of abstraction. We can, for instance, prove optimality of schedules in a setting where the duration of certain tasks is only known approximately, or we may prove that certain "bad" states can not be reached even when the timing of some events is not known exactly.

Whereas the *Verifier* has an option to compute the fastest execution leading to a certain state, there is no corresponding option to compute the slowest execution. Nevertheless, we can compute this slowest execution via a few successive approximations. Uppaal can conform our guess that after 200 time units all jobs will be completed, that is, that the following property is satisfied:

```
A[] now>=200 imply
         (Belt.end && Jobber1.begin && Jobber2.begin)
```

We even have

```
A[] now>=150 imply
         (Belt.end && Jobber1.begin && Jobber2.begin)
```

but not

```
A[] now>=110 imply
         (Belt.end && Jobber1.begin && Jobber2.begin)
```

So the jobbers need at most between 110 and 150 time units to complete all the jobs. By doing a binary search and reducing the size of the interval step by step, we may infer that the following property is satisfied:

```
A[] now>=127 imply
         (Belt.end && Jobber1.begin && Jobber2.begin)
```

but the next property is not:

```
A[] now>=126 imply
         (Belt.end && Jobber1.begin && Jobber2.begin)
```

Hence, the diagnostic trace for the last property gives the slowest possible schedule, which takes exactly 126 time units. In this schedule, the final transitions from the jobbers back to the begin state are missing, but these transitions take no time. So even when the jobbers always start working on jobs as soon as they can, and immediately grab tools whenever they become available, the worst case schedule is still 26 time units slower than the fastest schedule from Figure 1.16. Figure 1.18 visualizes the slowest schedule as a Gannt chart. At time 0, `Jobber1` grabs the first

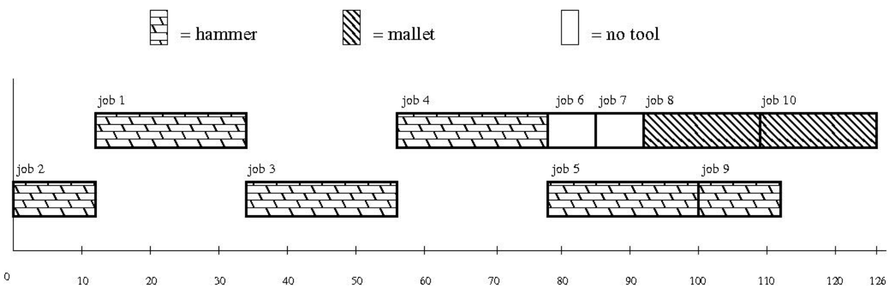


Fig. 1.18 Slowest schedule for completing the 10 jobs.

(hard) job, and `Jobber2` grabs the second (average) job. Of course, in any reasonable scenario, `Jobber1` would use the hammer and `Jobber2` the mallet. But in the worst case scenario of Figure 1.18, `Jobber2` gets the hammer, and hence `Jobber1` has to wait. Until time 78 one jobber is using the hammer while the other jobber is idling. At the end of the schedule, after `Jobber2` has finished job 9, `Jobber2` is still working for some time on job 10 using the mallet.

1.9 Parameters and arrays

In this section, we will discuss two useful features of Uppaal, that allow us to describe our jobshop model more compactly: template parameters and arrays.

1.9.1 Parameters

In the jobshop model, we have defined a single template `Jobber` with two instances `Jobber1` and `Jobber2`. Likewise, we would like to have a single template `Tool` with two instances `Hammer` and `Mallet`. It is possible to define this in Uppaal using the notion of *Parameters*. Parameters can be declared to have either call-by-value or call-by-reference semantics, that is, a template may have access to either a local copy of the argument or to the original. The syntax is taken from C++, where the identifier of a call-by-reference parameter is prefixed with an "&" in the parameter declaration. Clocks and channels must always be call-by-reference parameters.

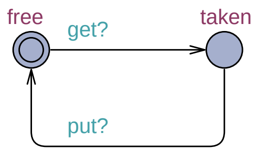


Fig. 1.19 Generic model of template `Tool`.

Figure 1.19 shows the definition of the generic template `Tool`. In the field *Parameters* at the top of the *Editor* tab, we declare the two channel names that are used as parameters in this template:

```
urgent chan &get, chan &put
```

In the *System declarations* section, we can now define `Hammer` and `Mallet` as instances of `Tool`:

```
Hammer = Tool(get_hammer,put_hammer);
Mallet = Tool(get_mallet,put_mallet);
```

The old templates `Mallet` and `Hammer` can be deleted via the *Remove template* option in the *Edit* menu. Our new model has exactly the same behavior (in terms of global states and transitions) as the old model, but due to the use of parameters the definition has become shorter.

1.9.2 Arrays

The model of the belt of Figure 1.14 is not easy to extend or reuse: each time we want to study a new arrival pattern of jobs, we have to redraw the automaton. This is cumbersome, especially if we want to schedule a batch with a large number (say hundreds) of jobs. It is also more natural to define the arrival pattern as a *data* *structure* rather than as a *control structure* (automaton). We can describe the job arrival pattern as a (constant) integer array as follows:

```
const int E=0;
const int A=1;
const int H=2;

const int J=10;
const int[0,2] jobs[J] = {H,A,H,H,H,E,E,A,A,A};
```

Uppaal does not support enumerated types and therefore we use 0 to encode easy jobs (E), 1 to encode average jobs (A), and 2 to encode hard jobs (H). As before, the integer constant J denotes the total number of jobs in the batch. The one dimensional constant array jobs specifies the arrival pattern of jobs (which is identical to the pattern in Figure 1.14). The size of the array equals J (counting from 0 to J − 1) and the range is the set $\{0,1,2\}$ or equivalently $\{\mathtt{E},\mathtt{A},\mathtt{H}\}$. Figure 1.20 shows a generic model for the Belt template that uses the information from the array jobs to generate the appropriate sequence of jobE!, jobA! and jobH! transitions. The automaton uses an auxiliary variable

```
int[0,J] i;
```

that records the number of jobs that has been delivered thus far. Again, the new model has exactly the same behavior (in terms of global states and transitions) as the old model, but due to the use of arrays the definition has become shorter and easier to reuse.

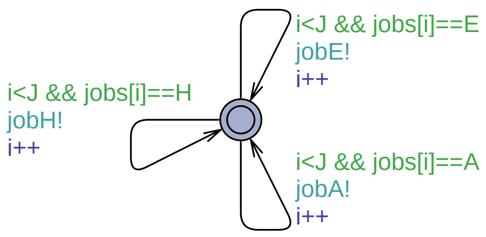


Fig. 1.20 Generic model of template Belt.

1.10 What is a good model?

After having presented the basic functionality of the Uppaal tool, we want to conclude this chapter with some general recommendations for constructing models. To some extent, building good models is an art. Dijkstra’s motto “Beauty is our business” [5] applies to models as well as to programs. Nevertheless, we can state seven criteria for good models.[6] These criteria are in some sense obvious, and any person with experience in modelling will often try to adhere to them. Often some criteria are hard to meet and typically several of them are conflicting. In practice, a good model is often one which constitutes the best possible compromise, given the current state-of-the-art of tools for modelling and analysis.

1. A good model has a clearly specified **object of modelling**, that is, it is clear what thing the model describes. The object of modelling can be (a part of) an existing artefact or physical system, but the object may also be a document that informally specifies a system or class of systems (for instance a protocol standard), and it may even be a collection of ideas of a design team about a system they construct, expressed orally and/or by some drawings on a whiteboard.
   In the case of our jobshop example, the object of modelling is the informal specification that is contained in the grey boxes throughout this chapter.
2. A good model has a clearly specified **purpose** and (ideally) contributes to the realization of that purpose. Possible purposes include: communication between stakeholders about a design, a specification of a system, verification of specific properties (safety, liveness, timing,..), analysis and design space exploration, code generation, and test generation. A model can be descriptive or prescriptive. If a model has to serve several distinct purposes then often it is better to construct multiple models rather than one.
   The only purpose of the jobshop models constructed in Sections 1.3 up to 1.7 is to explain the use of the Uppaal tool. The models in Section 1.8 serve the additional purpose that they help us to answer the timing related questions stated in the grey boxes.
3. A good model is **traceable**: each structural element of a model either (1) corresponds to an aspect of the object of modelling, or (2) encodes some implicit domain knowledge, or (3) encodes some explicit additional assumption. Additional assumptions are for instance required when a protocol standard is incomplete (e.g., it does not specify how to handle certain events in certain cases). Links between the structural elements of the model and the aspects of the object of modelling should be clearly documented. A distinction must always be made between properties of (a component of) a model and assumptions about the behavior of its environment.
   Our jobshop models are traceable: in the text we explain for each element in a model how it relates to the informal specification.
4. A good model is **truthful** (or **valid**): relevant properties of the model should also carry over to (hold for) the object of modelling. Typically, for each (relevant) behavior of the object of modelling there should be a corresponding behavior of the model. In the construction of models often idealizations or simplifications are necessary in order to allow for the use of a certain modeling formalism or in order to be able to analyze the model. In these cases, the model may not be entirely truthful. The modeller should always be explicit about such idealiza-

[6] Most of these criteria are described by Mader, Wupper and Boon [11]. We refer to [11] for further links to related work in the areas of software engineering, requirements analysis, and design.

tions/simplifications, and have an argument why the properties of the idealized model still say something about the artefact. In the case of quantitative models this argument will typically involve some error margin. In the case of timed automata models it frequently occurs that a model "overapproximates" reality and that, due to nondeterminism, certain behaviors that are possible in the model are not possible for the artefact.
The untimed model of Section 1.7 is certainly truthful to the informal specification of Milner listed in the grey box in Section 1.2. However, in the timed model of Section 1.8 there is at least one idealization that is not entirely realistic. Our model assumes that it takes no time to grab a job from the conveyor belt. Or more precisely: that no time elapses between starting to pick a job from the belt and starting to pick a tool. Since a job consists of both a peg and a block, it is reasonable to assume that a jobber needs both hands to grab a job from the conveyor belt. Hence it is not possible to grab a job and a tool simultaneously, and if we really want our model to be thruthful, we should add extra transitions that correspond to the end of the activity to grab a job, and we should give lower and upper bounds for the duration of this activity. In the jobshop example the criterion of thruthfulness collides with our next criterion of simplicity: in order to keep our model simple we compromised a bit on thruthfulness.

5. A good model is **simple** (but not too simple). Occam's razor is a principle particularly relevant to modelling: among models with roughly equal predictive power, the simplest one is the most desirable. Hence, the number of states and state variables should be as small as possible, and the level of atomicity of transitions should be as coarse grained as possible (but not coarser), i.e., the number of transitions should be minimal given the intended use of the model. Preferably, things should be written only once, and one should avoid ugly encodings. Preferably, the model uses stable, well-defined and well-understood concepts and semantics.
   The model of Section 1.9 is almost maximally simple. We would have liked to simplify the automaton of Figure 1.20 even further so that it only has one transition instead of three. But this is not possible since Uppaal does not allow data parameters for synchronization channels.
6. A good model is **extensible and reusable**, that is, it has been designed to evolve and be used beyond its original purpose. Typically, if one defines models in a modular and parametric way this allows for dimensioning, future extensions and modifications, especially if modules have well-defined interfaces. Ideally, a model should not just describe the specific system at hand: by appropriate instantiation and dimensioning it should be possible to model a whole class of similar systems.
   Our jobshop model can be extended or reused in several ways: we can easily increase the number of jobbers and tools, modify the sequence of jobs, and modify the timing parameters. What can not be changed so easily are the assumptions on which tools can be used for which jobs. The next chapter [] presents a more generic version of the jobshop model in which variations of these assumptions can be trivially modified. However, this requires the use of some advanced modelling features, which are explained in [].

7. A good model has been designed and encoded for **interoperability and sharing** of semantics. Model-driven development of an embedded system typically leads to a plethora of models, all presenting different views on and abstractions of the system. If a model is not somehow linked to other models, its usefulness will be limited. Ideally therefore, the relationships between all models should be properly defined, for instance via formal refinement relations.
   In this chapter, we have not addressed issues of interoperability and sharing. We refer to chapters XX and YY **add refs** of this handbook for a description of various links between Uppaal and other model based development tools. In a more realistic version of the jobshop example, in which for instance the jobbers are replaced by robots, one of the things one could do is to take the schedules computed by Uppaal and translate these to control programs for the robots. Such an approach (using Uppaal) is described for instance in [9]. **(and Chapter XX?)**

Clearly, there are many relationships and dependencies between the criteria. If a model is traceable, that is, links between the structural elements of the model and the aspects of the object of modelling are clearly documented, then chances increase that the model will be truthful. Also, if a model has been set up in a modular way, then one may apply a divide-and-conquer strategy both for establishing truthfulness of the model and for analysis.

biosection??

References

1. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, Cambridge, Massachusetts, 2008.
2. G. Behrmann, A. David, and K.G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Bertinoro, Italy, September 13-18, 2004, Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.
3. E.M. Clarke, E.A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.
4. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.
5. W.H.J. Feijen, Gasteren A.J.M. van, D. Gries, and J. Misra, editors. *Beauty is our business — A Birthday Salute to Edsger W. Dijkstra*, Texts and Monographs in Computer Science. Springer-Verlag, 1990.
6. O. Grumberg and H. Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*. Springer, 2008.
7. K. Havelund, A. Skou, K.G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using Uppaal. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 3-5, 1997, San Francisco, CA, USA*, pages 2–13. IEEE Computer Society, 1997.
8. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, 1985.
9. T. Hune, K.G. Larsen, and P. Pettersson. Guided synthesis of control programs using Uppaal. *Nord. J. Comput.*, 8(1):43–64, 2001.

10. C.A.J. Hurkens. Spreading gossip efficiently. *Nieuw Archief voor Wiskunde*, 5/1(2):208–210, June 2000.
11. A. Mader, H. Wupper, and M. Boon. The construction of verification models for embedded systems. Technical Report TR-CTIT-07-02, Centre for Telematics and Information Technology, University of Twente, The Netherlands, 2007.
12. R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs, 1989.
13. Wikipedia. List of model checking tools, November 2011. `http://en.wikipedia.org/wiki/List_of_model_checking_tools`.