1. UPPAAL and SCSI Protocol Example

By : André Wong

Abstract

Formal methods that are easy to use but with limited flexibility are sometimes criticized as inefficient and ineffective. This report took a detailed look at UPPAAL[1] [1], a toolbox for modeling, simulating and verifying real-time systems, that is jointly developed by Uppsala University and Aalborg University. Specifically, components of the UPPAAL toolbox, the modeling language and the verification language are investigated. A Small Computer System Interface (SCSI) bus protocol example is presented to illustrate various features of UPPAAL. In addition, there are discussions regarding advantages and tradeoffs in using UPPAAL.

It is concluded that UPPAAL is still in its infant research state and is currently suitable for modeling small real-time systems only. In general, formal methods should be able to model a reasonable variety of systems. Even if they are designed for restricted application domains, they should be flexible enough to satisfy common needs in those domains to be qualified as useful formal methods. It is apparent that UPPAAL does not meet the expectations of a useful formal method although a general conclusion cannot be drawn, as UPPAAL just happens to be a bad example that is in its early development stage. The author still believes that there is a need for formal methods that are user-friendly and the goal can be accomplished.

1. http://www.docs.uu.se/docs/rtmv/uppaal

Table of Contents

List of Figures

1. Introduction

In terms of user-friendliness, formal methods can be roughly divided into two categories - those that are easy to use, intuitive but are limited in verification capability and others that are mathematically-based and can be rigorously verified. Without a doubt, people have different definitions for the terms user-friendliness and intuitiveness. However, for the purpose of this report, a formal method is said to be user-friendly if it is suitable for typical programmers and software designers, i.e. people that have university education in computer science related fields but may not have deep mathematical backgrounds.

While some people argue that the ease of use of formal methods does not matter as users of those methods ought to be experts themselves, this is certainly not true in all cases. Since the number of experts in the formal methods is limited, it is not always economically viable to have software experts for every single software project, except for safety critical software like the Darlington nuclear plant shutdown cooling system. Besides, for large projects, ease of use of formal methods speeds up the development of specifications. Given the current rapid pace of technology, speed is important because software projects themselves may become obsolete before the specifications are finalized. Thus, complex formal methods usually do not tend to be scalable, either practically or economically. Ease of use also encourages the wider use of formal methods on all software projects, even if those methods are only loosely formal, since having loosely formal specifications is always better than having no formal specifications at all.

Although ease of use plays a role in the success of formal methods, it compromises on the expressiveness, flexibility and verifiability which are also vital. Intuitively, one would expect that the effect of usability is negligible on small systems but becomes substantial as the projects grow. While the exact balance is controversial and situation-related, this project will try to investigate one end of the spectrum.

A formal method called UPPAAL that claims to be easy to use and efficient in terms of computational resources is investigated to see what specifiers need to comprise in using user-friendly formal methods. However, UPPAAL is by no means the only research effort in this area. According to surveys in formal methods [7, 8], there are many other graphical formal methods like Statecharts [9], Petri Nets [10] and Timed Transition Models/Real-Time Temporal Logic [11]. UPPAAL is chosen among them because its syntax and semantics seem to be the simplest of all, and because it is freely available.

UPPAAL is a toolbox for modeling, simulating and verifying real-time systems which includes a modeling language, a simulator and a model checker. The modeling language is suitable for systems with non-deterministic processes with finite control structures and real value clocks. The simulator lets users simulate the models visually to explore their dynamic behavior. The model checker verifies safety, bounded-liveness and other user-specified properties by reachability analysis.

Section 2 of this report investigates the syntax and the semantics of the UPPAAL modeling language. Section 3 briefly introduces the UPPAAL simulator. Section 4 discusses the verification language and the model checker in details. Section 5 describes how the SCSI protocol works and how it can be modeled in UPPAAL. Sections 6 presents comments from the author and section 7 states the conclusion.

2. Modeling Language

The UPPAAL modeling language describes real-time systems as networks of timed automata. An interesting feature of the modeling language is that one can model the same system in either graphical or textual formats.

Figure 1 and 2 show a real-time system modeled in these respective formats. A graphical editor called Autograph$_1$ is used for drawing the system shown in Figure 1. The graphical format has the advantage of user-friendliness, while the textual format gives model specifiers a fast and convenient way to express complex systems. The UPPAAL modeling language can also be used to model linear hybrid automata [4], which are timed automata with clocks whose rates are bounded in a certain interval. Both formats are equivalent except when modeling linear hybrid automata, in which the textual format must be used. There are tools to convert a graphical model to a textual model for compilation purpose but not vice versa.

In the modeling language, a real-time system consists of a finite number of non-deterministic finite state automata. In addition, there is a finite number of states in each automaton, and each state can optionally

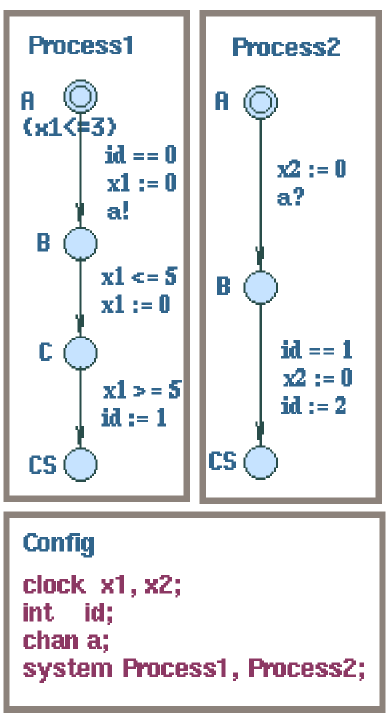

Figure 1. An example UPPAAL model

```
clock  x1, x2;
int    id;
chan a;

process Process1 {
  state  A {x1<=3}, B, C, CS;
  init   A;
  trans  A -> B {
           guard  id == 0;          sync   a!;            assign x1 := 0;
         },
         B -> C {
           guard  x1 <= 5;          assign x1 := 0;
         },
         C -> CS {
           guard  x1 >= 5;          assign id := 1;
         };
}

process Process2 {
  state  A, B, CS;
  init   A;
  trans  A -> B {
           sync   a?;               assign x2 := 0;
         },
         B -> CS {
           guard  id == 1;          assign x2 := 0,        id := 2;
         };
}

system Process1, Process2;
```

Figure 2. Same model in textual format

1. http://www.inria.fr/meije/verification/quick-guide.html/#Section4

contain a list of invariants. Transitions between any two states may be labeled with guard, synchronization, and assignment statements. The whole system contains global clock variables, data variables and synchronization channels that are visible to all automata. The graphical syntax of UPPAAL is introduced by illustrating an example.

Figure 1 shows a real-time system with two automata, with four and three states, respectively. Circles represent states, and the double circles indicate initial states. Expressions enclosed in brackets near the state names (i.e. $x1 \leq 3$) are invariants. Transitions between states are represented by black arrows that can be labeled with guard, synchronization or assignment statements. All gray boxes except the one with the title *Config* represent automata. The gray box with the title *Config* declares global clock variables, data variables, channels and automata in the system.

Since Figure 2 shows the same model as Figure 1 in the textual format, it is explained by drawing similiarites between the figures. Each automaton in Figure 1 corresponds to one *process* in Figure 2. A *process* statement consists of a state declaration list, an initial state statement and a state transition list. A state declaration list declares all states inside an automaton. Optionally, each state name can be followed by a list of invariants enclosed in a bracket representing invariants for that state. The state declaration list looks like

state stateName(invariant, invariant ...), stateName(invariant, invariant, ...), ...

In addition, the initial state of each automaton is declared by using the initial state statement :

init stateName

Each transition specifies the original and the final states of the transition. Similar to a transition in Figure 1, a transition in Figure 2 contains a list of guard statements, synchronization and assignment statements. A transition takes the form

```
stateName → stateName
{
    guard statements
    ...
    synchronization statements
    ...
    assignment statements
    ...
}
```

Finally, All transitions inside an automaton are declared in a transition list which looks like

trans transition, transition, transition ...

Clock variables, data variables, channels and automata declarations shown in the gray *Config* box in Figure 1 are declared at the top and the bottom of Figure 2.

The simplified textual syntax is presented here informally for further understanding. Consult [1] for the complete BNF syntax.

```
<global variable declarations>

process <automaton name>
{
        state <state declaration list>
        init <initial state>
        trans [ <start state> ] -> <end state>
        {
                [<guard list>]
                [<synchronization list>]
                [<assignment list>]
        },
        [ <start state> ] -> <end state>
        {
                [<guard list>]
                [<synchronization list>]
                [<assignment list>]
        },
        ...
}

<other automaton definition >
...

<automaton declarations>
```

Figure 3. Simplified informal syntax of the modeling language

The semantics of UPPAAL modeling language is similar to that of a conventional finite state machine. A UPPAAL model is a parallel composition of all of its automata. All automata start at their initial states and execute independently of each other unless synchronized via channels.

A transition is enabled when the conjunction of all of its guard statements is evaluated to be true and all the synchronization statements are executed, i.e. the channels specified by them are all synchronized. For each automaton, if one of the transitions that start from the current state is enabled, the current state of the automaton changes to the final state of that transition. If more than one transitions are enabled, one of them is chosen non-deterministically. Each transition consumes an unknown but finite amount of time that can be bounded by the use of invariants. Transitions of each automaton are assumed to happen indefinitely often or deadlock is said to occur.

Many of UPPAAL terms introduced in the preceding paragraph are explained in details in the following sections.

2.1 Global Variables

All variables are either data variables or clock variables. Since they are all global, they are shared variables that can be used by all automata.

Clock variables are similar to normal integer variables except their values are always non-negative. During each system time tick, all clocks increment by one automatically. As an example, two clock variables *x1* and *x2* are declared in Figure 1 using *clock* x1, x2.

In addition, a different kind of clock variables whose rates are bounded in an interval can be defined. In other words, if one defines a clock variable whose rate is specified by a lower bound *l* and an upper bound *u*, the clock variable increments by an amount between *l* and *u* during each system time tick. This kind of clock is useful in describing systems with clock rates that are known but with tolerance values. As an example, a clock variable *x3* with a lower bound 3 and an upper bound 5 can be defined by *clock* x3 *rate [3,5]*. As previously mentioned, clocks with bounded rates can only be used in the textual format.

For data variables, only integer data variables are supported in the current version of UPPAAL. As an example, an integer variable *id* is declared in Figure 1 using *int id*. Although this limitation restricts the flexibility of the modeling language, there are already plans to enhance support for other data types like records, floating point numbers and arrays to make UPPAAL closely resemble a high-level real-time programming language.

Synchronization channels are communication media that allow two automata to communicate with each other via signaling, i.e. channels do not carry data. For example, a channel *a* is declared in Figure 1 using *chan a*.

Similar to Calculus of Communicating Systems (CCS) [3], communication on channels occurs as a two-process synchronization, but the nature of this kind of synchronization prohibits broadcasting, which is common in many real-time systems. This problem is solved by introducing special states called committed locations.

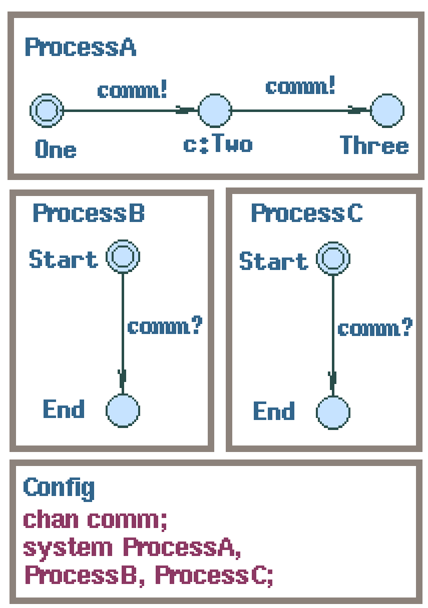

Figure 4. Broadcasting and committed location example

In Figure 4, *ProcessA* is to broadcast a signal to *ProcessB* and *ProcessC* via the channel *comm*. Normally, this is not a true broadcast since *ProcessA* synchronizes with *ProcessB* and *ProcessC* at different times. To ensure the atomicity of the broadcast (i.e. *ProcessA* synchronizes with ProcessB and ProcessC at the same time), the state *Two* of *ProcessA* is marked as a committed location as indicated by the *c:* prefix. The *c:* prefix makes state *Two* of *ProcessA* a temporary state that must be exited immediately and consumes no time, thus making synchronizations with *ProcessB* and *ProcessC* happen at the same time. Assume that at time 0, *ProcessA*, *ProcessB* and *ProcessC* are in states *One*, *Start* and *Start* respectively. Consider the case where the state *Two* of *ProcessA* is not a committed location. At time x, *ProcessA* reaches the state *Two*. Also, either *ProcessB* or *ProcessC* reaches the state *End,* and the other automata stays in the state *Start*, where x is a clock variable, and $x > 0$. At time y, *ProcessA* reaches the state *Three* and the automaton that stays in the state *Start* moves to the state *End*, where y is a clock variable and $y > x$. Since $x \neq y$, *ProcessB* and *ProcessC* are synchronized at the different times, thus broadcasting is not achieved. However, for the case where the state *Two* is a committed location, this scenario executes similarly with $y = x$. This means *ProcessB* and *ProcessC* are synchronized at the same time, and *ProcessA* passes through the state *Two* consuming no time.

Consequently, transitions following committed locations must contain only actions and no guard or channel reading statements, since committed locations must be exited immediately.

2.2 Invariants

Invariants are conditions expressing constraints on the clock variables in order for an automaton to remain in a particular state. If the invariant for a state is not true, and none of its transitions can be taken, UPPAAL reports an error. An invariant example is illustrated in Figure 1. $x1 \leq 3$ in state *A* of automaton *Process1* is an invariant. It ensures that *Process1* moves to state *B* before *x1* becomes larger than 3.

2.3 Guard Statements

Guard statements express conditions on the values of clock and data variables that must be satisfied for a transition to be taken. When there are more than one guard statements for a transition, the conjunction of all guard statements is evaluated. The default guard of a transition is the boolean value true. A guard statement example is illustrated in Figure 1. *id == 1* in the second transition of *Process2* guards the second transition. As a result, *Process2* cannot move to the state *CS* until *id* becomes 1.

2.4 Assignment Statements

Assignment statements in the UPPAAL modeling language are restricted forms of those found in other programming languages. Clock variable assignments must be of the form $x := n$, where n is a non-negative integer constant. Data variable assignments must be of the form $i := c*i + c'$, where c and c' are integer constants. Note that only the constants and the variable itself can appear on the right hand side of the assignment expression. An assignment statement example is illustrated in Figure 1. $x2 := 0$ in the first transition of automaton *Process2* is an assignment statement. It assigns 0 to the variable *x2*.

2.5 Synchronization Statements

Synchronizations are based on complementary actions of sending and receiving, denoted by *comm*! and *comm*? respectively, where *comm* is the channel name. All sends and receives are blocking. In other words, absence of synchronization action indicates an implicit self-looping similar to a τ-transition in CCS. A synchronization statement example is illustrated in Figure 1. *a!* in the first transition of the automaton *Process1* is a channel sending statement and *a?* in the first transition of the automaton *Process2* is a channel receiving statement. *a!* and *a?* together ensure that *Process1* and *Process2* reach their respective state *B* at the same time.

2.6 Delayed Transitions

State transitions are not guaranteed to happen the moment the transitions are enabled, as long as no invariants in the current state are violated. Using the system from Figure 1 as an example, *Process1* can leave the state *A* anytime from time 0 to time 3 even though the guard statement of the first transition is already determined to be true. However, this property may create problems for cases where a precise synchronization is required. A special type of channels called urgent channels can be used to ensure that when two processes are ready to synchronize, there will be no delays. Using the same example, *Process1* and *Process2* are guaranteed to synchronize at time 0 instead of anytime from time 0 to time 3 if they are synchronized through an urgent channel. Urgent channels can be declared by using the keyword *urgent chan* instead of *chan* for normal channels.

3. Simulator

The simulator in UPPAAL provides a way to visually simulate the dynamic behavior of the models built using the UPPAAL modeling language. Its main purpose is for debugging and gaining more in-depth understanding of models built. In order to symbolically simulate the system, the graphical form of UPPAAL modeling language must be used when building models. Since there are no tools to convert textual models to their graphical equivalents as stated before, textual models cannot be simulated.

The simulator is illustrated here for completeness but will not be further discussed, as it is not the focus of this report. Figure 5 shows a screen shot of several simulator windows.

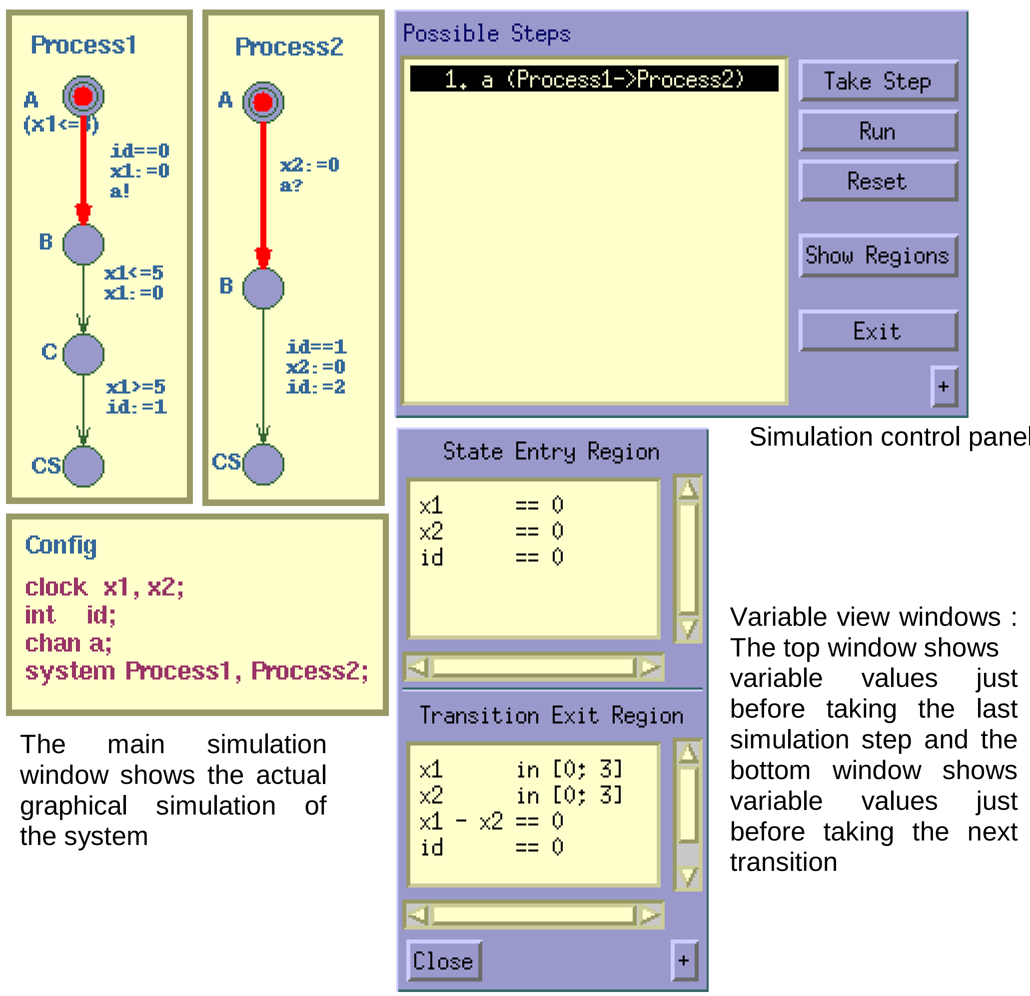

Figure 5. UPPAAL simulator simulates the example from Figure 1

4. Model Checker

UPPAAL models can be verified by specifying the desired properties using the UPPAAL verification language. The verification language is syntactically independent of the modeling language and is read separately by the UPPAAL model checker.

When models are verified, properties like invariants, absence of deadlocks and livelocks are automatically checked. In addition, users can also specify additional properties of the form

$$A[\,]\beta \mid E\langle\rangle\beta, \quad \beta ::= atomic \mid \beta_1 \ relation\ \beta_2 \mid not\ \beta \mid (\ \beta\ )$$

where *atomic* is an atomic formula being either a clock or a data constraint, or a proposition about a particular automaton in a particular state and *relation* is one of *or*, *and*, *imply*, >, ≥, <, ≤ and ==. Figure 6 shows some examples of user-specified properties.

A[]not (Process1.A and Process2.B)	It never happens that Process1 is in the state A while Process2 is in the state B
A[] (Process1.B imply Process2.B)	Whenever Process1 is in the state B, Process2 is also in the state B
E<> (Process1.CS and x1 ≥ 5)	There will eventually be an instance where Process1 is in the state CS and the clock x1 is greater than 5
E<> (Process2.CS)	Process2 will eventually reach the state CS

Figure 6. Example of user specified properties

The current version of UPPAAL model checker checks for invariant and reachability properties. Note that invariant and reachability properties can be translated to universal and existential formulae respectively. Since $A[\,]\beta = \sim E\langle\rangle \sim\beta$, those two properties are actually some negated versions of each other. In other words, the model checker actually only checks existential properties as seen from the algorithm that is presented in Figure 8.

It may be noted that the verification language is not flexible enough to specify many general real-time system properties. For example, consider the following bounded liveness property :

$$\Psi = atomic1\ until_{<t}\ atomic2$$

The above property states that the *atomic2* must hold before time *t* and *atomic1* must hold until then, where *atomic1* and *atomic2* are atomic formulae. For properties that cannot be expressed directly in the verification language, a workaround suggested by [2] is to manually create additional automata that observe changes to variables in the actual model, i.e *atomic1* and *atomic2* in this case. Those test automata reflect their observations by moving to different states. A test automaton for Ψ is illustrated in Figure 7.

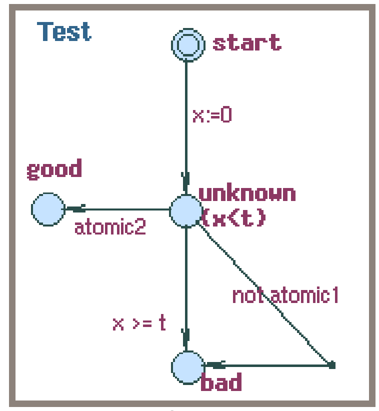

Figure 7. Test Automaton

The test automaton, *Test*, is actually a finite state machine that checks Ψ's validity. The three states *bad*, *good* and *unknown* indicate whether Ψ is valid, invalid and undetermined respectively. If it is possible for Ψ to be valid, *atomic1* is required to remain true before *atomic2* becomes true. Otherwise, the transition with the guard *not atomic1* is enabled, and *Test* moves to the state *bad*. If *atomic2* becomes true when x is less than t, the transition between the states *unknown* and *good* is enabled, and *Test* moves to the state *good*. If the previously event does not occur before x becomes greater or equal than t, the transition with the guard $x \geq t$ is enabled, and *Test* moves to the state *bad*. Finally, to actually prove Ψ, an extra property λ=*A[] not(Test.bad)* is needed to indirectly prove the bounded liveness property. The property states that the test automaton never stays in the state *bad*.

The manual addition of a test automaton may seem unintuitive but [2] showed that a system *S* satisfies Ψ if the parallel composition of the *S* and *Test* satisfies λ. Ψ is a bounded liveness property in this example but in general, it can be any arbitrary property.

The model checking algorithm in UPPAAL is based on an interpretation using a finite-state symbolic semantics of networks. More precisely, UPPAAL interprets the logic with respect to symbolic states of the form *(sv,d)*, where *d* is a constraint system (i.e. a conjunction of clock and data constraints) and *sv* is a state vector. Thus, *(sv,d)* represents states whose variable values satisfy constraint *d*. $\mapsto$ denotes the next executable successor relations between states. The symbolic algorithm shown in Figure 8 reduces the reachability problem to solving simple constraints systems. It checks whether an automaton can reach a state satisfying an existential quantified formulae β (i.e. E<>β). Universally quantified formulae are negated to existentially quantified formulae for verification purposes.

```
passed := { }
waiting := { (sv0,d0) }
repeat
    begin
        remove (sv,d) from waiting
        if (sv,d) satisfies β then return success
        else if not (d ⊆ d') for all (sv,d') ∈ passed then
        begin
            add (sv,d) to passed
            successor := { (svs,ds) : (sv,d) ⟼ (svs,ds) ^ ds ≠ 0 }
            for all (svs',ds') in successor do
                put (svs',ds') to waiting
        end
    end
until waiting == { }
return fail
```

Figure 8. The UPPAAL symbolic reachability algorithm

In Figure 8, the *waiting* set contains a set of *(sv,d)* that need to be checked against β, and the *passed* set contains a set of *(sv,d)* that have already been checked. Initially, the *waiting* set contains only (sv_0,d_0), where sv_0 is a state vector containing initial states of all automata and d_0 is the initial constraint system. The algorithm begins by removing an item *(sv,d)* from the *waiting* set and terminates with success if that *(sv,d)* satisfies β. Otherwise, *(sv,d)* is added to the *passed* set if *d* is not included in any of previously checked constrains systems. All successors (i.e. executable next state) of *(sv,d)* are explored and put in the *waiting* set, with the condition that constraint systems of those successors must be non-empty. Then another *(sv,d)* is removed from the *waiting* set again and this process repeats until the *waiting* set becomes empty. If the set becomes empty before a *(sv,d)* that satisfies β is found, the algorithm terminates with failure. Note that if queues are used instead of sets in the algorithm, the algorithm then executes in a breadth-first manner, which produces the shortest path to the goal. The shortest path can be used as a diagnose trace for the UPPAAL simulator. Additional details of the model checking algorithm can be found in [1].

The algorithm searches all executable states. It can be observed that the algorithm depends heavily on the $d \subseteq d'$ inclusion test and the emptiness test of d_s. [2] claims that these tests have been implemented efficiently by representing the constraint systems as weighted directed graphs, and can be evaluated in $O(n^3)$ where n is the number of variables in the constraint systems. The author has conducted some informal experiments and found out that UPPAAL is indeed quite efficient when compared to other formal method tools like SPIN [12].

5. A UPPAAL Example

The SCSI protocol, a relatively well-known protocol for connecting microcomputer devices to a common peripheral bus, is presented as an example to illustrate various features of UPPAAL. The SCSI protocol example is based on the description found in [5]. Using the SCSI protocol, multiple CPU's can share one or more SCSI buses, and each SCSI bus can daisy chain up to eight devices. SCSI is also widely used in PC servers by virtue of its ability to offload CPU utilization in data transfer. Because of the multiple sharing nature of SCSI buses, some kind of protocol is needed to coordinate bus arbitration and utilization. Figure 9 shows a typical SCSI bus configuration.

Devices on the SCSI bus are divided into two categories : initiators and targets. Initiators are devices that issue commands to others to perform certain operations, while targets are devices that accept commands from initiators and provide required services. Some devices can act as both initiators and targets.

The SCSI protocol proceeds in states called phases. Figure 10 shows various phases in the protocol. When the SCSI bus is idle, it is said to be in the *bus-free* phase. When some devices intend to seize control of the bus, they enter the *arbitration* phase to select a winner that will have the final control of the bus in this session. The winner then enters the *selection* or the *reselection* phase to select another device with which it wishes to exchange data. Subsequently, they enter the *information transfer* phase where data or control information can be transferred between them freely.

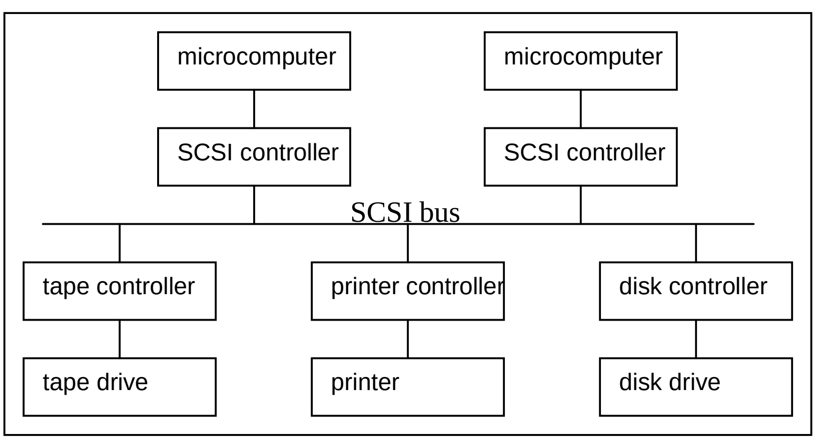

Figure 9. Example use of SCSI bus

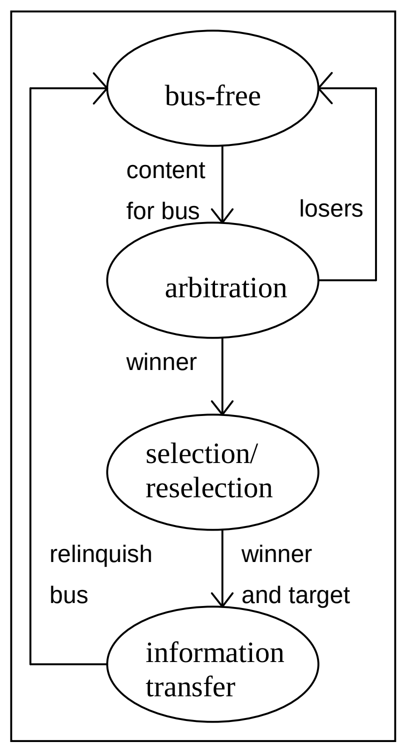

Figure 10. Phases of SCSI bus

When the target relinquishes control of the bus, the bus returns to the *bus-free* phase.

The SCSI bus has around eighteen signals. Only some of the relevant signals (shown below) will be used in the explanation of the SCSI protocol.

Name	Description
DB0-DB7	There are eight data lines, and each of them is associated with one device on the SCSI bus. The data lines are used during arbitration and selection phases. In the arbitration phase, a device activates its associated data line when a device wishes to identify itself. During the selection phase, a device activates the data line of the target device in order to select the target.
BSY	the bus is busy
SEL	a device is trying to select other devices; BSY and SEL together determine which phase the SCSI bus is in

There are eight data lines in the SCSI bus (*DB0-DB7*), and each has an associated ID ranging from 0 to 7, corresponding to each of the eight devices on the bus. Those ID's determine priority of the devices in the bus arbitration with 7 being the highest priority and 0 the lowest. Throughout this section, *DBx* is defined as the associated data line for device *x*, where *x* ranges from 0 to 7.

5.1 Arbitration

The SCSI arbitration is a distributed scheme in which all devices participate. A device can request the bus only when the bus is free, indicated by *BSY* and *SEL* being simultaneously inactive for more than 800 nanoseconds (ns). To request the bus, a device activates *BSY* and its associated data line at the same time. Once *BSY* is activated, no other device can request the bus. However, there is still a need for bus contention because several devices may already be waiting for the bus when it is being used. Once the bus is free, it is possible that those devices all activate *BSY* and their data lines at the same time.

2200 ns after all devices activate their data lines, they examine all data lines and determine the same winner independently. Those who lose, put their data lines back in inactive state within 800 ns. The winner activates *SEL* and waits for 1200 ns to allow time for others to deactivate their data lines. The role of the winner may be an initiator or a target. An initiator enters the *selection* phase to start a new I/O operation, while a target enters the *reselection* phase to resume a previously suspended operation.

Since a priority scheme instead of round robin is used, starvation is possible. Thus, devices should not be connected arbitrarily to the SCSI bus but according to the actual priority needs.

5.2 Selection

To select a target, the initiator activates *SEL*, the data line of the target, and its own data line which should have been already activated. After 80 ns, it deactivates *BSY* to indicate the start of the *selection* phase. All devices wait for 400 ns and then examine their associated data lines to check if they have been selected. The selected target responds by activating *BSY*. The initiator releases *SEL* and data lines it has activated previously to indicate the end of the *selection* phase.

5.3 Reselection

The reselection feature of SCSI bus prevents devices from tying up the bus unnecessarily. Imagine that a fast computer gains control of the SCSI bus and issues a read command to a tape drive. However, the data needed from the tape drive may not arrive until seconds later. To make efficient use of the SCSI bus, the computer relinquishes control of the bus after issuing the read command. When the data from the tape drive is ready, the tape drive gains control of the bus and enters the *reselection* phase to transfer the data back to the computer.

The *reselection* phase is essentially the same as the *selection* phase with the role of initiators and targets interchanged. A target device will try to gain control of the bus and transfer data to the initiator. Everything is the same up to the point where the initiator answers back with the *BSY* activated. Then the target also activates *BSY* before releasing *SEL*, and they enter the *information transfer* phase just like the *selection* phase.

5.4 Information Transfer

During the *information transfer* phase, the initiator and the target can transfer information freely. The exact detail of this phase will not be modeled as only simple data transfer is happening between the two devices. The actual data transfer is a trivial signal transfer that follows a *request data-data transfer-acknowledge* pattern, which does not add new requirements to the SCSI bus protocol.

BSY is activated throughout the *information transfer* phase. Deactivating *BSY* indicates the end of the *information transfer* phase and the bus is free again. Some sample timing diagrams of the SCSI protocol are presented in the Appendix to enhance the understanding of the SCSI protocol.

5.5 SCSI Protocol Model

The SCSI protocol explained previously is modeled using the UPPAAL modeling language, and the graphical model is presented in the Appendix. The model uses nearly all features of the UPPAAL modeling language to illustrate their functionality.

The example is nearly the same as described earlier : there are eight devices in the system in addition to the initialization process that set all variables and all devices to their initial settings. Eight devices then compete for, gain, and relinquish control of the bus indefinitely. Although every effort has been tried to produce a model as accurate as possible, the model is still far from perfect for the following reasons :

- As previously mentioned, the actual data transfer in the *information transfer* phase is not modeled, as the actual data transfer is trivial and does not add new requirements to the SCSI bus protocol.
- To reduce cluttering of the diagram, a device number *x,* where *x* ranges from 0 to 7, always selects a target device *x+1 mod 8*. It is trivial to modify the model so that a device can randomly select any other target device but at the expense of a much larger diagram. To have arbitrary device selection, 8 extra states are needed for each potential target device. This adds additional 384 states to the whole model, which increases the overall size by more than four times. Moreover, this simplification should not affect the correctness of the SCSI protocol.
- The reselection of initiators is performed randomly rather than according to the history. Historical information about the initiators should be stored by the target device, and is not part of the SCSI specification. Therefore, the model has no information about the original initiators during the reselection.
- Physical properties like electrical interference and propagational delay due to cable length cannot be modeled in software. The SCSI protocol model is only as correct as the assumptions about the environment.

The following properties are used to verify correctness of the SCSI protocol model. The actual properties written in the UPPAAL verification language are presented in the Appendix.

- there can only be a maximum of one initiator at any time
- it is possible for any device to grab the bus
- in the *selection* phase, an initiator device selects a target device and the target device answers back to the initiator device. Therefore, it is impossible for a target device to answer the selection if no initiator device exists

- in the *reselection* phase, a target device reselects an initiator device and the initiator device answers back to the target device. Therefore, it is impossible for an initiator device to answer the reselection if no target device exists
- 80 ns after leaving the *information transfer* phase, all devices must be in the idle state, indicated by *BSY*=0 and SE*L*=0
- all devices must wait for *SEL*=0 and *BSY*=0 for 800 ns before entering the *arbitration* phase
- when two devices are transferring data, they must be in the *information transfer* phase, indicated by *SEL*=1 and *BSY*=0
- when a device is in the *selection* state and before another device answers, *SEL*=0 and *BSY*=1
- when a device is in the *arbitration* phase, *SEL*=1 and *BSY*=0

After the SCSI protocol was modeled in UPPAAL, it was surprising to find out that no major problems or limitations were reported. Many components in the protocol naturally translate to constructs in the UPPAAL modeling language. For instance, timing constraints in the protocol become invariants, and the concurrency inherited in UPPAAL converts each SCSI device to an automaton. Electrical wires like *BSY* and *SEL* are modeled as integer variables. Since the SCSI protocol does not hold any other kind of data, no data variables are needed. Although it seems like the protocol can be trivially modeled in UPPAAL, the size of the example may be too small to justify any favorable claim.

However, upon careful inspection, it can been seen that the protocol is by no means a typical system, but rather a system in a specialized domain. If one treats *BSY*, *SEL* and other electrical wires as signals, the protocol model does not need any data variable at all, as it merely describes how the system reacts to changes in signal values. The size of the example is small enough that modularity does not become an issue. If one were to model a large system that is data-dependent, it is not clear how such model can be built using the UPPAAL modeling language even if it is a real time system. It seems that UPPAAL is only suitable for modeling small real-time systems that are not data-dependent, whether it can be used to model other kinds of systems effectively it not clear. Unfortunately, the answer is probably a negative one.

6. Comments about UPPAAL

As guessed from its syntax, UPPAAL model language is small, simple and remarkably easy to use. The graphical interface of the UPPAAL toolbox also puts its user-friendliness years ahead of many other formal method research tools. The intuitive modeling and verification languages also decrease the slope of the learning curve significantly. UPPAAL lives up to its claim in terms of user-friendliness.

In the SCSI protocol example, there are eight devices each having fifteen states, in addition to the ten states of the initialization process. Although the model has close to twenty-five billion total possible states, verification of properties seems to take seconds, and the memory requirement is negligible. This is considered extremely efficient when some other formal method tools like SPIN [12] take hundreds of megabytes of memory even when verifying small systems. The main reason for the efficient verification is the application of the symbolic techniques that reduces verification problems to that of manipulating and solving simple constraint systems, as discussed in section 4. Restrictions to reachability analysis are also crucial to the efficiency of the model checkers [13].

For non real-time systems, data variable constraints and temporal logic properties can usually represent most desired properties. However, for real-time systems, relaxed temporal properties like temporal logic are insufficient as precise timing constraints are needed for their correctness. Being a formal method designed for real-time systems, UPPAAL's inclusion of timing information in the modeling and the verification languages is preferable considering that many other formal methods do not provide this feature.

Nevertheless, all these advantages come with huge tradeoffs. UPPAAL modeling language lacks the flexibility and expressiveness of other formal methods. The inflexibility comes from the limited variety of models can that be modeled effectively. As discussed in the previous section, UPPAAL seems to be suitable for modeling small real time systems. In additional, even though [1] stated that one of the goals of UPPAAL is to make the modeling language as close to a high level real-time programming language as possible, many simple programming constructs are not implemented in its current version. Even the functionality of the assignment statement is severely restricted. Consequently, any models that are slightly data-related cannot be modeled effectively. In addition, UPPAAL modeling language has no support for modeling in a modular manner. All automata in a system must be modeled at the same level, i.e. no hierarchical support. The use of global variables, which is considered a bad practice by today's programming standard, is necessary in the language. All these limitations cause problems in large projects. Researchers of UPPAAL realize that these limitations hinder wide uses of UPPAAL and are planning to add supports for constructs like records, floating point numbers and arrays.

The limited expressiveness of UPPAAL comes from its verification language. Many temporal logic properties cannot be expressed directly or easily using the verification language. Properties like bounded-liveness need to be checked with the construction of additional test automata, placing extra burden on the model specifiers. By contrast, SPIN provides automatic translation of linear temporal logic (LTL) properties to *never* claims, letting model specifiers reason about properties at a more generic level instead of at the actual verification language level. Those *never* claims are actually Büchi automata that are generated mechanically based on a simple on-the-fly construction [14]. They are generated in a way that they formally accept only those system executions that satisfy the corresponding LTL formulae. Therefore, the inclusion of those generated automata in a SPIN model lets SPIN indirectly verify the LTL formulae, much like what test automata do for UPPAAL. However, the virtue of SPIN comes from the encapsulation of the LTL formulae conversion.

In fact, given the current simple algorithm UPPAAL uses for verification, substantial upgrades to the verification engine are needed to support verification of complex properties. A simpler solution is to provide a facility to automatically convert temporal logic properties to the corresponding test automata in a manner like SPIN, which [1] says will be included in the next version of UPPAAL.

The numerous feature comparisons of SPIN with UPPAAL in this section are not incidental. The similarity between SPIN and UPPAAL has drawn some attentions among researchers [6]. Compared to UPPAAL, SPIN has a much richer feature set and a more complete modeling language but lacks timing support. Although it provides a better simulation facility, the less than ideal verification tool is unable to verify large models. Some concurrency actions like broadcasting cannot be modeled accurately. However, unlike UPPAAL, it allows LTL formulae to be automatically translated to Büchi automata for verification.

7. Conclusion

UPPAAL is an easy to use formal method that is suitable for modeling systems with real-time constraints. Concurrent systems that are not data-dependent will find UPPAAL appealing. Intuitive tools and modeling language also make UPPAL ideal for beginners in the formal method realm.

However, large systems or systems that are data-dependent or need rigorous verification will find UPPAAL unattractive, partly due to its early research status and UPPAAL designers' desires to keep semantics of UPPAAL simple.

UPPAAL is an interesting and usable tool if its researchers can eliminate many of the limitations. At this moment, those limitations severely affect the quality and the type of models that can be built. Until then, UPPAAL is not a recommended tool for any real project. Nevertheless, the promise of UPPAAL indicates that it is entirely possible for formal methods to be intuitive enough to overcome compromises on the verifiability and expressiveness.

References

1. Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson and Wang Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, 22-24 October, 1995.
2. Kim G. Larsen, Paul Pettersson and Wang Yi. UPPAAL in a Nutshell. Springer International Journal of Software Tools for Technology Transfer 1(1/2), October 1997.
3. R. Milner. Communication and Concurrency. Prentice Hall, Englewood Cliffs, 1989.
4. A. Olivero, J. Sifakis, and S. Yovine. Using Abstractions for the Verification of Linear Hybrids Systems. In Proc. of CAV '94, volume 818 of Lectures Notes in Computer Science, 1994.
5. Zvonko G. Vranesic. Safwat G. Zaky. Microcomputer Structures, Orlando: Saunders College Publishing, 1989.
6. Henrik Ejersbo Jensen, Kim G. Larsen, and Arne Skou. Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL. In Proceedings of the 2nd SPIN Workshop. Rutgers University, New Jersey, USA, 5 August 1996.
7. Constance Heitmeyer and Dino Mandrioli, "Formal Methods for Real-Time Computing: An Overview," in Constance Heitmeyer and Dino Mandrioli (eds), in Formal Methods for Real-Time Computing. John Wiley, New York, 1996.
8. J.S. Ostroff. Formal Methods for the Specification and Design of Real-Time Safety Critical Systems. Journal of Systems and Software, Vol. 18, No. 1, April 1992.
9. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and M. Trachtenbrot. Statemate: a working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering, 16:403-414, 1990.
10. W. Reisig. Petri nets: an introduction. Springer-Verlag, Berlin, 1985.
11. J.S. Ostroff. Real-time computer control of discrete event systems modelled by extended state machines: a temporal logic approach. Technical Report 8618, Systems Control Group, Dept. of Electrical Engineering, University of Toronto, September 1986. revised January 1987.
12. Gerard J. Holzmann. Design and Validation of Computer Protocols, Prentice Hall, 1991.
13. Kim G. Larsen, Paul Pettersson and Wang Yi. Model-Checking for Real-Time Systems. In Proc. of Fundamentals of Computation Theory, volume 965 of Lecture Notes in Computer Science, pages 62-88, August 1995.
14. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-The-Fly Automatic Verification of Linear Temporal Logic. Proc. IFIP/WG6.1 Symp. Protocol Specifications, Testing and Verification (PSTV95), pp. 3-18, Warsaw, Poland, Chapman & Hall, June 1995.