Model Validation for Embedded Systems Using Formal Method-Aided Simulation

Daniel Karlsson, Petru Eles, Zebo Peng

ESLAB, Dept. of Computer and Information Science, Linköpings universitet

{danka, petel, zebpe}@ida.liu.se

Abstract

Embedded systems are becoming increasingly common in our everyday lives. As technology progresses, these systems become more and more complex. At the same time, the systems must fulfill strict requirements on reliability and correctness. Informal validation techniques, such as simulation, suffer from the fact that they only examine a small fraction of the state space. Therefore, simulation results cannot be 100% guaranteed. Formal techniques, on the other hand, suffer from state space explosion and might not be practical for huge, complex systems due to memory and time limitations. This paper proposes a validation approach, based on simulation, which addresses some of the above problems. Formal methods, in particular model checking, are used to aid, or guide, the simulation process in certain situations in order to boost coverage. The invocation frequency of the model checker is dynamically controlled by estimating certain parameters related to the simulation speed of the particular system at hand. These estimations are based on statistical data collected during the validation session, in order to minimise verification time, and at the same time, achieve reasonable coverage.

1 Introduction

It is a well-known fact that we increasingly often interact with electronic devices in our everyday lives. Such electronic devices are for instance cell phones, PDAs and portable music devices like Mp3-players. Moreover, other, traditionally mechanical, devices are becoming more and more computerised. Examples of such devices are cars, aeroplanes or washing machines. Many of them are also highly safety critical such as aeroplanes or medical equipment.

It is both very error-prone and time-consuming to design such complex systems. At the same time, there is a strong economical incentive to decrease the time-to-market.

In order to manage the design complexity and to decrease the development time, designers usually resort to reusing existing components (so called IP blocks) so that they do not have to develop certain functionality themselves from scratch. These components are either developed in-house by the same company or acquired from specialised IP vendors [1], [2].

Designers using IP blocks in their complex designs must be able to trust that the functionality promised by the IP providers is indeed implemented by the IP block. For this reason, the IP providers must thoroughly validate their blocks. This can be done

either using formal methods, such as model checking, or using informal methods, such as simulation.

Both methods, in principle, compare a model of the design with a set of properties (assertions), expressed in a temporal logic (for instance (T)CTL), and answer whether they are satisfied or not. With formal methods, this answer is mathematically proven and guaranteed. The state space induced by the model under verification (MUV) is exhaustively investigated by applying techniques for efficient state space representation [3] and efficient pruning of the same. However, using informal methods, this is not the case. Such methods are not exhaustive and therefore require a metrics which captures the reliability of the result. Such metrics are usually referred to as *coverage metrics* [4], as they try to express how big part of the state space has been covered by the simulation. Unfortunately, formal methods such as, for example, model checking, suffer from state space explosion. Although there exist methods to relieve this problem [5], [6], [7], for very big systems simulation-based techniques are needed as a complement. Simulation techniques, however, are also very time consuming, especially if high degrees of coverage are required.

This paper presents a validation technique combining both simulation and model checking. The basis of the approach is simulation, but model checking methods are applied to faster reach uncovered parts of the state space, thereby enhancing coverage. The invocation of the model checking is dynamically controlled during run-time in order to minimise the overall validation time.

This paper is organised in 12 sections and one appendix. Section 2 presents related work, followed by an overview of the proposed approach in Section 3. Section 4 introduces preliminary background material needed to fully understand the paper. The coverage metrics used throughout the paper and the concept of assertion activation are described in Section 5 and Section 6 respectively. Section 7 and Section 8 present the frameworks for stimulus generation and assertion generation, whereas the coverage enhancement procedure is explained in Section 9. Section 10 describes the dynamic stop criterion for the simulation, which also determines the invocation time of the model checker. Experimental results are found in Section 11, and Section 12 concludes the paper. Appendix A contains additional material on the generation of a PRES+ model given an ACTL formula.

2 Related Work

Combining formal and informal techniques is however not a new invention. Symbolic trajectory evaluation [8] is a technique that mixes model checking with symbolic simulation. Though it has proved to be very efficient on quite large designs, it can only be applied on a small class of properties.

Another idea, proposed by Tasiran et al., involves using simulation as a way to generate an abstraction of the simulated model [9]. This abstraction is then model checked. The output of the model checker serves as an input to the simulator in order to guide the process to uncovered areas of the state space. This will create a new abstraction to model check. If no abstraction can be generated, it is concluded that the specification does not hold. As opposed to this approach, the technique presented in this paper does

not iteratively model check a series of abstractions, but tries to maximise simulation coverage given a single model. There is hence a difference in emphasis. They speed up model checking using simulation, whereas this work improves simulation coverage using model checking.

Another approach, developed at Synopsys, uses simulation to find a “promising” initial state for model checking [10]. In this way, parts of the state space, judged to be critical to the specification, are thoroughly examined, whereas other parts are only skimmed. The approach is characterised by a series of partial model checking runs where the initial states are obtained with simulation. The technique presented in this paper is, on the other hand, coverage driven in the sense that model checking is used to enhance the coverage obtained by the simulation.

Shyam et al. [11] (with later elaboration by De Paula et al. [12]) propose a technique where formal verification techniques are used to guide the simulation towards a certain verification goal, e.g. coverage. The core idea is the use of a distance function that gives a measure on how far away the current simulation state is from the verification goal. The simulation is then guided based on minimising this distance on an abstracted model. Our approach differs in the sense that our MUV is not abstracted and the formal state evaluation is not applied at each simulation step, allowing the less computationally intensive simulation carry out as much work as possible. They perform frequent formal analysis on approximate models, whereas we perform the formal analysis more seldomly but on the actual model.

Lu et al. [13] focus on the fact that methods for formal verification of limited sized IP blocks at Register-Transfer Level (RTL) have previously been frequently researched, but less attention has been paid to verifying designs at the system-level. The idea is to use randomly generated legal traces obtained from the formal verification of the IP blocks to guide the simulation at system-level. In this way, the verification of the whole system will be more efficient resulting in higher quality simulation results. In our approach, on the other hand, formal verification guides the simulation at the same level of abstraction in the same verification round, thereby increasing the quality of simulation.

The work by Abts et al. [14] formally verifies the architectural model of the design. The trace obtained from the formal verification is then fed into the simulator in order to examine the particular scenario described by the trace, in detail. This approach is somehow opposite to Lu et al., but the difference to our work remains in principle the same.

Techniques where simulation guides formal verification have also been developed. The approach developed by Krohm et al. [15] uses simulation as a complement to BDD-based techniques for equivalence checking. Our technique applies to property checking, as opposed to equivalence checking. Moreover, formal verification is in our approach applied to boost simulation, as opposed to Krohm et al. who apply simulation to boost formal verification, though at the expense of accuracy.

Gupta et al. [16] apply formal methods to automatically generate a testbench for guided simulation given a model under veri-

fication and a CTL property. As a result of their approach, they obtain a testbench which generates stimuli for a subsequent simulation. The stimuli are biased towards the CTL property to increase the probability for obtaining a witness or counter-example. We have also included a similar type of biasing into our approach, though our biasing technique is not based on formal methods but on examining the property itself.

Simulation can also help in concretising counter examples from model checking. Nanshi et al. [17] have targeted the problem of lengthy counter examples through model abstraction. By performing simulation, the counterexample from the abstract model can be mapped to the corresponding concrete model.

Formal methods have also been used for test case generation, as in the technique developed by Hessel et al. [18]. The main idea is to apply model checking on a correct model of the system and extract test cases from the diagnostic trace. The test cases are then applied to the actual implementation. The approach is guided by a certain coverage metrics and the resulting test cases are guaranteed to minimise test time. Although their work bears certain similarities with the work presented in this paper, it solves a different problem. They assume that the model is correct while the implementation is to be tested. In our case, however, it is the model that has to be proven correct. Another version of this problem is addressed by Fey et al [19]. They use model checking to detect behaviour in the design that is not yet covered by the existing testbench.

As opposed to existing simulation-based methods, the presented approach is able to handle continuous time (as opposed to discrete clock ticks) both in the model under validation and in the assertions. It is moreover able to automatically generate the "monitors", which are used to survey the validation process, from assertions expressed in temporal logic. In addition to this, the method dynamically controls the invocation frequency of the formal verification tool (model checker), with the aim of minimising validation time while achieving reasonable coverage.

3 Methodology Overview

The objective of the proposed validation technique is to examine if a set of temporal logic properties, called assertions, are satisfied in the model under validation (MUV). The technique imposes the following three assumptions:

- The MUV is modelled as a transition system, e.g. PRES+ (see Section 4.2).
- Assertions, expressed in temporal logics, e.g. (T)CTL (see Section 4.1), stating important properties which the MUV must not violate, are provided.
- Assumptions, expressed in temporal logics, e.g. (T)CTL (see Section 4.1), stating the conditions under which the MUV shall function correctly (according to its assertions), are provided.

The assertions and assumptions described above constrain the behaviour on the interface of the MUV. They do not state anything about the internal state. The assumptions describe the valid behaviour of the environment of the MUV, i.e. constrain the input, while the assertions state what the MUV must guarantee, i.e. constrain the output. As mentioned previously, the objective of the validation is to examine if the MUV indeed satisfies its assertions.

The result of the verification is only valid to the extent expressed by the particular coverage metrics used. Therefore, certain measures are normally taken to improve the quality of the simulation with respect to the coverage metrics. This could involve finding corner cases which only rarely occur under normal conditions. Coverage enhancement is therefore an important part in simulation-based techniques.

The proposed strategy consists of two phases, as indicated in Figure 1: simulation and coverage enhancement. These two phases are iteratively and alternately executed. The simulation phase performs traditional simulation activities, such as transition firing and assertion checking. When a certain stop criterion is reached, the algorithm enters the second phase, coverage enhancement. The coverage enhancement phase identifies a part of the state space that has not yet been visited and guides the system to enter a state in that part of the state space. After that, the algorithm returns to the simulation phase. The two phases are alternately executed until the coverage enhancement phase is unable to find an unvisited part of the state space.

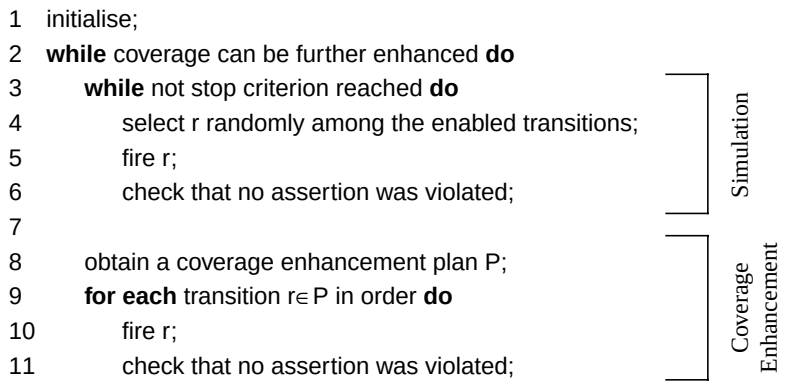

Figure 1. Verification Strategy Overview

In the simulation phase, transitions are repeatedly selected and fired at random, while checking that they do not violate any assertions (Line 4 to Line 6). This activity goes on until a certain stop criterion is reached (Line 3). The stop criterion used in this work is, in principle, a predetermined number of fired transitions without any coverage improvement. This stop criterion will be further elaborated in Section 10.

When the simulation phase has reached the stop criterion, the algorithm goes into the second phase where it tries to further enhance coverage by guiding the simulation into an uncovered part of the state space. An enhancement plan, consisting of a sequence of transitions, is obtained and executed while at each step checking that no assertions are violated (Line 8 to Line 11).

It is at this step, obtaining the coverage enhancement plan, that a model checker is invoked (Line 8).

The two phases, simulation and coverage enhancement, are iteratively executed until coverage is considered unable to be further enhanced (Line 2). This occurs when either 100% coverage has been obtained, or when the uncovered aspects, with respect to the coverage metrics in use, have been targeted by the coverage enhancement phase at least once, but failed.

Stimulus generation is not explicitly visible in this algorithm, but is covered by the random selection of enabled transitions (Line 4) or as part of the coverage enhancement plan (Line 8). Subsequent sections will go into more details about the different parts of the overall strategy.

4 Preliminaries

4.1 Computation Tree Logic (CTL)

Both the assertion to be checked and the assumptions under which the MUV is verified are, throughout this paper, expressed in a formal temporal logic with respect to the ports in the interfaces of the MUV. In this work, we use (timed) Computation Tree Logic, (T)CTL [20], [21] for expressing these properties. However, other similar logics may be used as well.

CTL is a branching time temporal logic. This means that it can be used for reasoning about the several different futures (computation paths) that the system can enter during its operation, depending on which actions it takes. CTL formulas consist of path quantifiers (**A**, **E**), temporal quantifiers (**X**, **G**, **F**, **U**, **R**) and state expressions. Path quantifiers express whether the subsequent formula must hold in all possible futures (**A**), no matter which action is taken, or whether there must exist at least one possible future where the subformula holds (**E**). Temporal quantifiers express the future behaviour along a certain computation path (future), such as whether a property holds in the next computation step (**X**), in all subsequent steps (**G**), or in some step in the future (**F**). p **U** q expresses that p must hold in all computation steps until q holds. q must moreover hold some time in the future. q **R** p on the other hand expresses that p must hold in every computation step until q, or if q never holds, p holds indefinitely. State expressions may either be a boolean expression or recursively a CTL formula.

In TCTL, relations on time may be added as subscripts on the temporal operators (except the **X** operator). For instance, $\mathbf{AF}_{\leq 5}\, p$ means that p must hold before 5 time units and $\mathbf{AG}_{\leq 5}\, p$ means that p holds continuously during at least 5 time units.

An important sublogic is (T)ACTL. A (T)CTL formula belongs to (T)ACTL if it only contains universal path quantifiers (**A**) and negation only occurs in front of atomic propositions. This type of properties express, in particular, safety and liveness. The sublogic is, in the context of this paper, important when it comes to stimulus generation and assertion checking (Section 7 and Section 8 respectively).

4.2 The design representation: PRES+

In the discussion throughout this paper, as well as in the tool we have implemented, the MUV, the assertion checker and stimulus generator are assumed to be modelled in (or translated into) a design representation called *Petri-net based Representation for Embedded Systems* (PRES+) [22]. This representation is chosen over other representations, such as timed automata [23], due to its intuitivity in capturing important design features of embedded systems, such as concurrency and real-time aspects. The representation is based on Petri-nets with some extensions as defined below. Figure 2 shows an example of a PRES+ model.

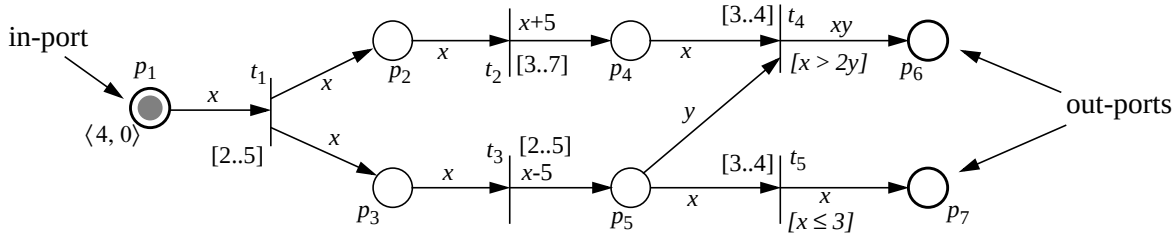

Figure 2. A simple PRES+ net

Definition 1. PRES+ model. A PRES+ model is a 5-tuple $\Gamma = \langle P, T, I, O, M_0 \rangle$ where

P is a finite non-empty set of *places*,

T is a finite non-empty set of *transitions*,

$I \subseteq P \times T$ is a finite non-empty set of *input arcs* which define the flow relation from places to transitions,

$O \subseteq T \times P$ is a finite non-empty set of *output arcs* which define the flow relation from transitions to places, and

M_0 is the initial *marking* of the net (see Item 2 in the list below).

We denote the set of places of a PRES+ model Γ as $P(\Gamma)$, and the set of transitions as $T(\Gamma)$. The following notions of classical Petri-nets and extensions typical to PRES+ are the most important in the context of this work.

1. A token k has values and timestamps, $k = \langle v, r \rangle$ where v is the value and r is the timestamp. In Figure 2, the token in place p_1 has the value 4 and the timestamp 0. When the timestamp is of no significance in a certain context, it will often be omitted from the figures.
2. A marking M is an assignment of tokens to places of the net. The marking of a place P is denoted $M(p)$. A place p is said to be marked iff $M(p) \neq \varnothing$.
3. A transition t has a function (f_t) and a time delay interval ($[d_t^- .. d_t^+]$) associated to it. When a transition fires, the value of the new token is computed by the function, using the values of the tokens which enabled the transition as arguments. The timestamp of the new tokens is the maximum timestamp of the enabling tokens increased by an arbitrary value from the time delay interval. The transition must fire at a time before the one indicated by the upper bound of its time delay interval (d_t^+),

but not earlier than what is indicated by the lower bound (d_t^-). The time is counted from the moment the transition became enabled. In Figure 2, the functions are marked on the outgoing edges from the transitions and the time interval is indicated in connection with each transition.

4. The PRES+ net is forced to be safe, i.e. one place can at most accommodate one token. A token in an output place of a transition disables the transition.
5. The transitions may have guards (g_t). A transition can only be enabled if the value of its guard is true (see transitions t_4 and t_5).
6. The preset ${}^\circ t$ (postset t°) of a transition t is the set of all places from which there are arcs to (from) transition t. Similar definitions can be formulated for the preset (postset) of places. In Figure 2, ${}^\circ t_4 = \{p_4, p_5\}$ and $t_4{}^\circ = \{p_6\}$.
7. A transition t is enabled (may fire) iff there is one token in each input place of t, no token in any output place of t and the guard of t is satisfied.

We will now define a few concepts related to the component-based nature of our methodology, in the context of the PRES+ notation.

Definition 2. Union. The union of two PRES+ models $\Gamma_1 = \langle P_1, T_1, I_1, O_1, M_{01}\rangle$ and $\Gamma_2 = \langle P_2, T_2, I_2, O_2, M_{02}\rangle$ is defined as $\Gamma_1 \cup \Gamma_2 = \langle P_1 \cup P_2, T_1 \cup T_2, I_1 \cup I_2, O_1 \cup O_2, M_{01} \cup M_{02}\rangle$.

Other set theoretic operations, such as intersection and the subset relation, are defined in a similar manner.

Definition 3. Component. A component C is a subgraph of the graph of the whole system Γ such that:

1. Two components $C_1, C_2 \subseteq \Gamma$, $C_1 \neq C_2$, may only overlap with their ports (Definition 4), $P(C_1 \cap C_2) = P_{connect}$, where $P_{connect} = \{p \in P(\Gamma) | (p^\circ \subseteq T(C_2) \wedge {}^\circ p \subseteq T(C_1)) \vee (p^\circ \subseteq T(C_1) \wedge {}^\circ p \subseteq T(C_2))\}$.
2. The pre- and postsets (${}^\circ t$ and t°) of all transitions $t \in T(C)$ must be entirely contained within the component C, ${}^\circ t, t^\circ \subseteq P(C)$.

Definition 4. Port. A place p is an out-port of component C if $(p^\circ \cap T(C) = \varnothing) \wedge ({}^\circ p \subseteq T(C))$ or an in-port of C if $({}^\circ p \cap T(C) = \varnothing) \wedge (p^\circ \subseteq T(C))$. p is a port of C if it is either an in-port or an out-port of C.

Definition 5. Interface. An interface of component C is a set of ports $I = \{p_1, p_2, \ldots\}$ where $p_i \in P(C)$.

The PRES+ representation is not required by the verification methodology itself, although the algorithms presented here have been developed using PRES+. However, if another transition-based design representation is found more suitable, similar algorithms can be developed for that design representation. Our current implementation also accepts models specified in SystemC, which are compiled to PRES+ as described in [24].

In the context of PRES+, two forms of boolean state expressions are used within (T)CTL formulas: p and $p \Re v$. The expression p refers to a state where a token is present in place p. Expressions of the form $p \Re v$, on the other hand, describe states where also the value of a token is constrained. The value of the token in place p (there must exist such a token) must relate to the value v as indicated by relation $\Re$. For example, the expression $p = 10$ states that there is a token in place p, and that the value of that token is equal to 10.

In order to perform the model checking, the PRES+ model, as well as the (T)CTL properties, have to be translated into the input language of the particular model checker used. For the work in this paper, the models are translated into timed automata [23] for the UPPAAL model checking environment [25], using the algorithms described by Cortés et al. [22]. The properties are also modified to refer to timed automata elements, rather than PRES+. The model checker was invoked with default parameters, such as breadth first search and no approximation.

4.3 Example

The model in Figure 3, the assumption in Equation (1) and the assertion in Equation (2) will serve as an example throughout this paper, in order to clarify the details of the methodology. The assumption (Equation (1)), under which condition the validation will take place, stipulates that only even numbers will appear in p, whereas the assertion (Equation (2)), which eventually will be verified, states that if p contains a token with a value less than 20, then a token will appear in q (regardless the value) within 10 time units.

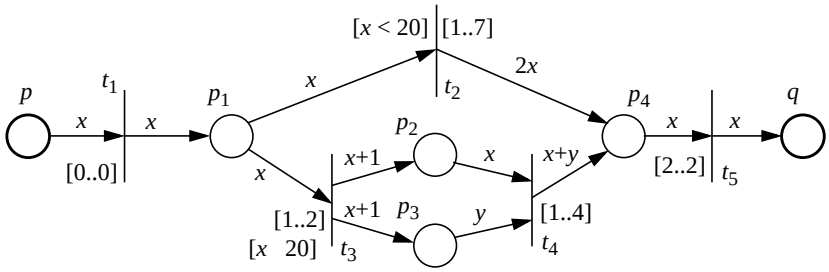

Figure 3. An explanatory PRES+ model

$$\mathbf{AG}(p \rightarrow even(p)) \tag{1}$$

$$\mathbf{AG}\ (p < 20 \rightarrow \mathbf{AF}_{\leq 10} q) \tag{2}$$

5 Coverage Metrics

Coverage is an important issue in simulation-based methods. It provides a measure of how thorough a particular validation is. It is advantageous to use a coverage metrics which refers both to the implementation and specification, so that the validation process exercises all relevant aspects related to both. A combination of two coverage metrics is therefore used throughout this paper:

assertion coverage and transition coverage.

Definition 6. Assertion coverage. The assertion coverage (cov_a) is the percentage of assertions which have been activated (activation will be defined in Section 6) during the validation process (a_{act}) with respect to the total number of assertions (a_{tot}), as formalised in Equation (3).

$$cov_a = \frac{a_{act}}{a_{tot}} \tag{3}$$

Definition 7. Transition coverage. The transition coverage is the percentage of fired distinct transitions (tr_{fir}) with respect to the total number of transitions (tr_{tot}), as formalised in Equation (4).

$$cov_{tr} = \frac{tr_{fir}}{tr_{tot}} \tag{4}$$

Definition 8. Total coverage. The total coverage (*cov*) (*coverage* for short) is computed by dividing the sum of activated assertions and fired transitions by the sum of the total number of assertions and transitions, as shown in Equation (5).

$$cov = \frac{a_{act} + tr_{fir}}{a_{tot} + tr_{tot}} \tag{5}$$

Assuming, in Figure 3, that for a particular validation session transitions t_1, t_2 and t_5 have been fired and the assertion in Equation (2) has been activated, the assertion, transition and total coverage are 100%, 60% and 67% as computed in Equation (6), Equation (7) and Equation (8) respectively.

$$cov_a = \frac{1}{1} = 1 \tag{6}$$

$$cov_{tr} = \frac{3}{5} = 0.6 \tag{7}$$

$$cov = \frac{3+1}{5+1} \approx 0.67 \tag{8}$$

6 Assertion Activation

During simulation, a record of fired transitions and activated assertions has to be kept, in order to compute the achieved coverage. As for recording fired transitions, the procedure is straight-forward: For each transition fired, if it has never been fired before, add it to the record of fired transitions. However, when it comes to assertions, the procedure is not as obvious. Intuitively, an assertion is activated when all (relevant) aspects of it have been observed or detected during simulation. In order to formally

define the *activation* of an assertion, the concept of *assertion activation sequence* needs to be introduced.

The purpose of an activation sequence is to provide a description of what markings have to occur before the assertion is considered activated. As will be demonstrated shortly, the order between some markings does not matter, whereas it does between other markings. For this reason, an activation sequence is not a pure sequence, but a partial sequence defined as a set of <number, markings>-pairs. The ordering between markings in pairs with the same order number is undetermined, whereas markings with different order numbers have to appear in the order indicated by the number.

The set of markings in a pair is denoted by a place name possibly augmented with a relation. This place name represents all markings with a token in that place, and whose value satisfies the relation. Below follows a formal definition of assertion activation sequence.

Definition 9. Assertion activation sequence. An assertion activation sequence is a set of pairs <*d*, *K*>, where *d* is an integer and *K* is a (T)CTL atomic proposition, representing a set of markings.

Equation (9), given below, shows an example of an activation sequence. The order between *p* and $q = 5$ is irrelevant. However, they must both appear before *r*. Proposition *p* stands for the set of all markings with a token in place *p*, the value of the token does not matter. Proposition $q = 5$ represents the set of all markings with a token in *q* with a value equal to 5. Lastly, proposition *r* denotes the markings with a token in place *r*.

$$\{\langle 1, p\rangle, \langle 1, q = 5\rangle, \langle 2, r\rangle\} \tag{9}$$

To use activation sequences for detecting assertion activations, a method to derive such a sequence given an assertion ((T)CTL formula) must be developed. In the following discussion, it is assumed that the assertion only contains the temporal operators **R** and **U**. Negation is only allowed in front of atomic propositions. Any formula φ can be transformed to satisfy these conditions. Moreover, time bounds on the temporal operators of TCTL formulas, e.g. <10 in $\mathbf{AF}_{<10}\, q$ are dropped, since, in the context of activation sequences, we are only interested in the ordering of markings, not in the particular time moment they occur. The following function, $A(\varphi)$ returns a set of activation sequences corresponding to the formula φ.

Definition 10. $A(\varphi)$. The function $A(\varphi) = A(\varphi, 0)$ returning a set of activation sequences given an ACTL formula is recursively defined as:

- $A(p, d) = \{\{\langle d, p\rangle\}\}$, $A(\neg p, d) = \{\{\langle d, \neg p\rangle\}\}$
- $A(p \mathfrak{R} v, d) = \{\{\langle d, p \mathfrak{R} v\rangle\}\}$

  $A(\neg(p \mathfrak{R} v), d) = A(\neg p \vee p \overline{\mathfrak{R}} v, d)$
- $A(false, d) = \varnothing$, $A(true, d) = \{\varnothing\}$

- $A(\varphi_1 \vee \varphi_2, d) = A(\varphi_1, d) \cup A(\varphi_2, d)$
- $A(\varphi_1 \wedge \varphi_2, d) = \bigcup_{a \in A(\varphi_1, d)} \bigcup_{b \in A(\varphi_2, d)} (a \cup b)$
- $A(\mathbf{Q}[\varphi_1 \mathbf{R} \varphi_2], d) = A(\varphi_1, d+1) \cup A(\neg\varphi_2, d+1)$
- $A(\mathbf{Q}[\varphi_1 \mathbf{U} \varphi_2], d) = A(\neg\varphi_1, d+1) \cup A(\varphi_2, d+1)$

It should be noted that $A(\varphi)$ returns a set of activation sequences. The interpretation of this is that if any of these sequences have been observed during simulation, the corresponding assertion is considered activated. Assume, for example, the set of activation sequences in Equation (10). It contains two sequences. The first sequence captures a situation where a token in *p* should appear before a token in *q*, whereas the second sequence captures the single occurrence of a token in *r*. Either of these two scenarios activates, according to this set of sequences, the corresponding assertion.

$$\{\{\langle 1, p\rangle, \langle 2, q\rangle\}, \{\langle 1, r\rangle\}\} \tag{10}$$

The function $A(\varphi)$ is moreover provided with an auxiliary parameter *d*, initially 0, in order to keep track of the ordering between the markings in the resulting sequence.

The activation sequence corresponding to $\varphi = p$, an atomic proposition, is the singleton sequence containing markings with a token in place *p*. Detecting a token in *p* is consequently sufficient for activating that property. Similarly $\varphi = \neg p$ and $\varphi = p \Re v$ are activated by markings where there is no token in *p* and where there is a token in *p* with a value satisfying the relation, respectively. $\varphi = \neg(p \Re v)$ is activated if there either is no token in *p* or the token value is outside the specified relation.

Since there is no marking which satisfies $\varphi = \mathit{false}$, there cannot exist any activating sequence. The formula $\varphi = \mathit{true}$, on the other hand, is activated by all markings. There is consequently no constraint on the marking. This situation is denoted by an empty sequence. As will be explained shortly, such a property will therefore be immediately marked as activated.

Disjunctions introduce several possibilities in which the property can be activated. It is partly for the sake of disjunctions that $A(\varphi)$ returns a set of sequences, rather than a single one. The function returns the union of the sequences of each individual disjunct. It is sufficient that one of these sequences is detected during simulation to consider the property activated.

In conjunctions, the activation sequences corresponding to both conjuncts must be observed. Since both conjuncts may correspond to several activation sequences, the two sets of sequences must be interleaved so that all possibilities (combinations) are represented in the result.

The formula $\mathbf{Q}[\varphi_1 \mathbf{R} \varphi_2]$, for any $\mathbf{Q} \in \{\mathbf{A}, \mathbf{E}\}$, is considered activated when either of the following two scenarios occurs:

1. After φ_1 is detected, then from the point of view of this property, the following observations are of no significance any more.

Therefore, detecting φ_1 is sufficient for activating this property.

2. A similar scenario applies when φ_2 no longer holds, therefore also $\neg\varphi_2$ is sufficient for activation.

Both scenarios refer to future markings, for which reason the order number (parameter *d*) is increased with 1.

The **U** operator follows a similar pattern as the **R** operator. An important characteristics of a $\mathbf{Q}[\varphi_1\ \mathbf{U}\ \varphi_2]$ formula, for any $\mathbf{Q} \in \{\mathbf{A}, \mathbf{E}\}$, is that φ_2 must appear in the future. The property does not specify anything about what should happen after φ_2. Therefore, φ_2 is considered sufficient for activating the property. Similarly, the property does not specify what should happen when φ_1 no longer holds. Detecting $\neg\varphi_1$ is therefore also sufficient for activating the property. Since both scenarios refer to the future, the order number (parameter *d*) is increased with 1.

It is important to distinguish the procedure of deriving assertion activation sequences from the verification process. The sequences only identify possible orderings in which the involved markings may occur to in order to satisfy the assertion. It is the task of the validation process to assess whether they are indeed satisfied or not.

Consider the example assertion in Equation (2), presented in a normalised form, and without the time constraints, in Equation (11).

$$\begin{array}{l}\mathbf{AG}(p < 20 \rightarrow \mathbf{AF}\ q) \Leftrightarrow \\ \mathbf{A}[\ false\ \mathbf{R}\ (\neg(p < 20) \vee \mathbf{A}[true\ \mathbf{U}\ q])]\end{array} \tag{11}$$

The set of activation sequences corresponding to this formula is computed as follows:

$$A(\mathbf{A}[\ false\ \mathbf{R}\ (\neg(p < 20) \vee \mathbf{A}[true\ \mathbf{U}\ q])], 0) =$$
$$A(\ false, 1) \cup A(\neg(\neg(p < 20) \vee \mathbf{A}[true\ \mathbf{U}\ q]), 1) =$$
$$\varnothing \cup A(p < 20 \wedge \mathbf{E}[\ false\ \mathbf{R}\ \neg q], 1) =$$
$$\bigcup_{a \in A(p < 20, 1)} \bigcup_{b \in A(\mathbf{E}[\ false\ \mathbf{R}\ \neg q], 1)} (a \cup b) =$$
$$\bigcup_{a \in \{\{\langle 1,\ p < 20\rangle\}\}} \bigcup_{b \in \{\{\langle 2,\ q\rangle\}\}} (a \cup b) = \{\{\langle 1,\ p < 20\rangle, \langle 2,\ q\rangle\}\}$$

with the following auxiliary computations:

$$A(p < 20, 1) = \{\{\langle 1,\ p < 20\rangle\}\}$$
$$A(\mathbf{E}[\ false\ \mathbf{R}\ \neg q], 1) = A(\ false, 2) \cup A(q, 2) = \{\{\langle 2,\ q\rangle\}\}$$

As can be seen in the computation, the activation sequence $\{\langle 1, p < 20\rangle, \langle 2, q\rangle\}$ is the only one activating $\mathbf{AG}(p < 20 \rightarrow \mathbf{AF}\ q)$. According to the sequence, a token in *p* with a value less than 20, which is eventually followed by a token in *q*, activates the assertion.

As will be seen in Section 7, activation sequences are not only used for computing the assertion coverage, but they are also useful for biasing the input stimuli to the MUV in order to boost assertion coverage.

7 Stimulus Generation

The task of stimulus generation is to provide the model under validation with input consistent with the assumptions under which the model is supposed to function. In the presented approach, the stimulus generator consists of another model, expressed in the same design representation as the MUV, i.e. PRES+. A more elaborate description on how to generate such a model, given an assumption ACTL formula, is presented in Appendix A. At this moment, it is just assumed that it is possible to derive such a PRES+ model corresponding to an ACTL formula. This model encodes all possible behaviours which a PRES+ model can perform without violating the property for which it was created.

The stimulus generator and the MUV are then connected to each other during simulation, by applying the union of the two models, to form a closed system. For this reason, the stimulus generator is not explicit in the pseudocode in Figure 1. An enabled transition selected on Line 4 might belong to the MUV as well as to the stimulus generator.

As mentioned previously, let us assume that only even numbers are accepted as input to port *p* in the model of Figure 3. This assumption was formally expressed in Equation (1), but is repeated for convenience in Equation (12). Following the discussion above, a model capturing this assumption is generated and attached to the MUV. The result is shown in Figure 4. A transition, which immediately consumes tokens, is attached to port *q*, implying the assumption that output on *q* must immediately be processed.

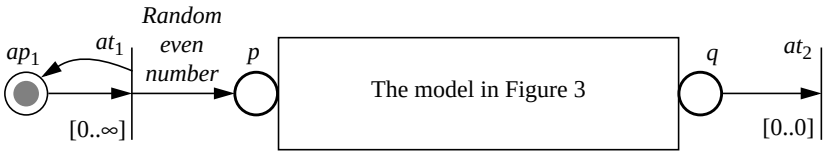

Figure 4. A MUV with stimulus generators

$$\mathbf{AG}(p \rightarrow even(p)) \tag{12}$$

It was mentioned previously that activation sequences can be used to boost assertion coverage during the simulation phase. This can be achieved by not letting the algorithm (Figure 1) select a transition to fire randomly (Line 4). The transition selection should be biased so that transitions leading to a marking in an activation sequence are selected with preference, thereby leading the validation process to activating one more assertion. When all markings in the sequence have been observed, the corresponding assertion is considered activated.

As shown in Section 6, $A(\varphi)$ translates a logic formula φ into a set of sequences of markings (represented by a place name, possibly augmented with a relation on token values). The transition selection algorithm should select an enabled transition which leads to a marking which is first (with lowest order number) in any of the sequences. However, only selecting transitions strictly

according to the activation sequences could lead the simulator into a part of the state space, from which it will never exit, leaving a big part of the state space unexplored. Therefore, an approach is proposed in which the transition selection is only guided by the activation sequences with a certain probability. The proposed transition selection algorithm is presented in Figure 5.

```
1  function selectTransition(MUV: PRES+,
       actseqs: set of activation sequences) returns transition
2      entrans := the set of enabled transitions in MUV;
3      p := random[0..1];
4      if p < p_c then
5          if ∃ t ∈ entrans,
               such that t leads to the first marking in any seq. in actseqs then
6              return t;
7      return any transition in entrans;
```

Figure 5. The transition selection process

A random value *p*, denoting a probability, is chosen between 0 and 1 (Line 3). If that value is less than a user-defined parameter p_c (Line 4), a transition is selected following an activation sequence if such transition exists (Line 5 and Line 6), otherwise a random enabled transition is selected (Line 7). The algorithm in Figure 5 is called on Line 4 in Figure 1.

The user-defined parameter p_c controls the probability to select a transition which fulfils the activation sequence. This value introduces a trade-off which the designer has to make. The lower the value of p_c is, the higher is the probability to enter unexplored parts of the state space. On the other hand, a too low value of p_c might lead to the situation where the assertions are rarely activated and, then, it could take longer time to achieve a high assertion coverage. In the experiments presented in Section 11, the value of p_c has been set to 0.66. With this value, the validation process is biased towards targeting events specified by the activation sequences while still giving substancial freedom for random exploration of the statespace.

8 Assertion Checking

The objective of validation is to ensure that the MUV satisfies certain desired properties, called assertions. The part of the simulation process handling this crucial issue is the assertion checker, also called monitor [26]. Designers often have to write such monitors manually, which is a very error-prone activity. The key point is to write a monitor which accepts the model behaviour if and only if the corresponding assertion is not violated.

A model created from an ACTL formula was introduced for stimulus generation in Section 7, based on the technique illustrated in Appendix A. The same type of models can also be used for assertion checking as monitors. The assertion in Equation (2) will be used to illustrate the operation of a monitor throughout this section. Figure 6 shows the essential part of the monitor corresponding to that assertion. For the sake of clarity, the ports *p* and *q* are omitted. Arrows connecting to these ports are labelled with

the name of the corresponding port.

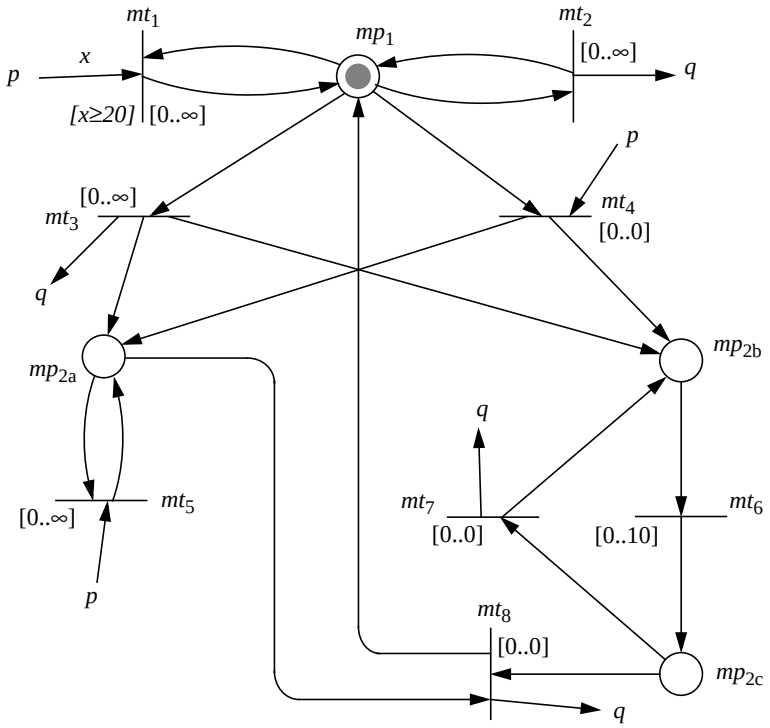

Figure 6. Part of an example monitor

The structure of monitors, generated with the technique illustrated in Appendix A, follows a certain pattern. All transitions in Figure 6, except one, interact directly with a port, either *p* or *q*. The exception is transition mt_6 (located in the lower right quarter in the figure), whose purpose is to ensure that a token is put in *q* before the deadline, 10 time units after $p<20$. Such transitions, watching a certain deadline, are called *timers*. This observation is important when analysing the output of the MUV.

Figure 7 illustrates the intuition behind assertion checking. Both the input given by the stimulus generator and the output from the MUV are fed into the assertion checker. The assertion checker then compares this input and output with the monitor model generated from the assertion (like the one in Figure 6). For satisfiability, there must exist a sequence of transitions in the monitor leading to the same output as provided by the MUV, given the same input. This method works based on the fact that the monitor model captures all possible interface behaviours satisfying the assertion, including the interface behaviour of the MUV. The essence is to find out whether the MUV behaviour is indeed included in that of the monitor.

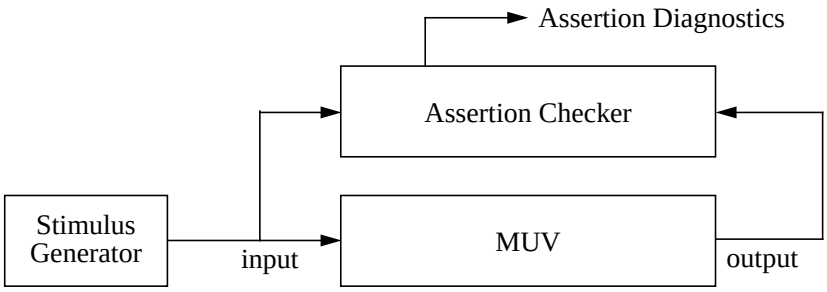

Figure 7. Assertion checking overview

As indicated in the figure, the input given to the MUV is also given to the assertion checker. That is everything that needs to be performed with respect to the input. As for the output sequence, on the other hand, the assertion checker has to perform a more complicated procedure. It has to find a sequence of transitions producing the same output.

It was mentioned previously, that all transitions are directly connected to a port. Due to this regularity, the stipulated output can always be produced by (at least) one of the enabled transitions in the monitor. If not, the assertion does not hold. The exception is timers. If, at the current marking, a timer is enabled, the timer is first (tentatively) fired before examining the enabled transitions in the same manner as just described. Successfully firing the timer signifies that the timing aspect of the assertion is correct.

Several enabled transitions may produce the same output. In the situation in Figure 6, for example, both transitions mt_2 and mt_3 are enabled and can produce the output q. However, firing either of them will lead to different markings and will constrain the assertion checker in future firings. The monitor has several possible markings where it can go, but the marking of the MUV only corresponds to one (or a few) of them. The problem is that the assertion checker cannot know which one will be followed. Therefore, the assertion checker has to maintain a *set* of possible current markings, rather than one single marking. The assertion checker, thus, has to go through each marking in the set and compare it with the marking on the interface of the MUV. If the marking currently being compared leads to a difference with the MUV, that marking is removed from the set. The assertion is found unsatisfied when the set of current markings is empty. Figure 8 presents the assertion checking algorithm. It replaces Line 6 and Line 11 in Figure 1. Line 1, Line 2 and Line 3 (Figure 8) are, however, part of the initialisation step at Line 1 in Figure 1. The algorithm uses the auxiliary function in Figure 9, which validates the timing behaviour of the model, and the function in Figure 10, which implements the output matching procedure.

```
1  monitor: PRES+ := model corresponding to the assertion to be checked;
2  curmarkings: set of markings := { initial marking of monitor };
3  newmarkings: set of markings;
4  ...
5      oldtime := current time in MUV;
6      fire r; -- Line 5 or Line 10 in Figure 1
7      newtime := current time in MUV;
8      curmarkings :=
           validateTimeDelay(newtime - oldtime, curmarkings, monitor);
9      if r provided MUV with an input then
10         put the tokens produced by r as input to
               each marking in curmarkings;
11     if r provided MUV with an output then
12         e := marking in the out-ports of MUV;
13         newmarkings := ∅;
14         for each m ∈ curmarkings do
15             set marking of monitor to m;
16             newmarkings := newmarkings ∪ findOutput(e, monitor);
17         curmarkings := newmarkings;
18     if curmarkings = ∅ then
19         abort; -- Assertion not satisfied
20 ...
```

Figure 8. The assertion checking algorithm in the context of Figure 1

```
1  function validateTimeDelay(d: delay, curmarkings: set of markings,
       monitor:PRES+) returns set of markings
2      newmarkings : set of markings := ∅;
3      for each m ∈ curmarkings do
4          set marking of monitor to m;
5          let time advance in monitor with d;
6          if not monitor exceeded the upper bound of the time delay interval
                of any enabled transition then
7              newmarkings := newmarkings ∪ { m };
8      return newmarkings;
```

Figure 9. Algorithm to check the timing aspect of an assertion

```
1  function findOutput(e: output marking, monitor: PRES+)
       returns set of markings
2      newmarkings : set of markings := ∅;
3      fire all enabled timers;
4      entrans := the set of enabled transitions in monitor;
5      initmarking := the current marking of monitor;
6      for each t ∈ entrans do
7          fire t in monitor;
8          if output marking of monitor = e then
9              if a timer has a token in its output place then
10                 move the token to the input place;
11             newmarkings := newmarkings ∪ { current marking of monitor };
12         set marking of monitor to initmarking;
13     return newmarkings;
```

Figure 10. Algorithm for finding monitor transitions fulfilling the expected output

Throughout the validation process, the simulator must maintain a global variable, on behalf of the assertion checking, containing the set of all possible current markings in the monitor. In Figure 8, the variable *curmarkings* is used for this purpose. The variable *newmarkings* is an auxiliary variable whose use will soon be explained.

The assertion checking algorithm must, at a certain moment, know how long (simulated) time a transition firing takes in order to detect timing faults. The variable *oldtime* contains the current time before the transition was fired and *newtime* the time after. The difference between the values of these two variables is the time it took for the transition, denoted *r*, to fire. This value is passed to the function validateTimeDelay (Figure 9) which validates the delay with respect to the assertion. The function examines all markings in *curmarkings* and returns the subset which still satisfies the assertion. The function will be explained in more detail shortly.

If the fired transition, *r*, provides an input to the MUV, that input is also added to each marking in *curmarkings*, so that the monitor is aware of the input (Line 9 and Line 10, Figure 8). As input counts either putting a token in an in-port of the MUV or consuming a token from an out-port.

If the fired transition, *r*, provides an output from the MUV (Line 11), that output must be compared with the monitor model in the assertion checker. As output counts either putting a token in an out-port of the MUV or consuming a token from an in-port.

Since the monitor potentially can be in any of the markings in *curmarkings*, all of these markings have to be examined (Line 14), one after the other. The monitor is first set to one of the possible current markings, after which the enabled transitions are examined with the function in Figure 10 (Line 16). The function returns a set of markings which successfully have produced the output. The members of this set are added to the auxiliary set *newmarkings*. Later, when all current markings have been examined, the new markings are accepted as the current markings (Line 17).

If, at this point, the set of current markings is empty, no monitor marking can produce the output and the assertion is concluded unsatisfied (Line 18 and Line 19).

The function in Figure 9 validates the timing aspects of an assertion. It examines the markings in *curmarkings* one after the other (Line 3). At each iteration, time is advanced in the monitor (Line 5). As a consequence, all enabled transitions are checked, so that the upper bound of their time delay interval is not exceeded (Line 6). If at least one transition exceeded its time bound, the marking currently under examination does not agree with the stipulated delay, and is skipped. Otherwise (Line 6), the marking is added to the result set *newmarkings* (Line 7) which later is returned (Line 8) and becomes the new set of current markings (Line 8 in Figure 8).

Let us now focus on the auxiliary function in Figure 10. Given an output marking (the marking in the out-ports of the MUV) and a monitor, the function returns the set of markings which satisfy the given output.

At this point, it can be assumed that the timing behaviour of the assertion has not been violated, since the function in Figure 9 was called prior to this one. Thus the timers do no longer play any roll. Because of this and the fact that the lower bound of the time delay interval of timers is 0, it is safe to fire all enabled timers (Line 3). At this moment, all enabled transitions are directly connected with a port. Each of these enabled transitions are fired one after the other (Line 6 and Line 7). The result after each firing is checked whether it matches the desired output, denoted *e* (Line 8). If it does, the new marking should be stored in *newmarkings* in order to later be returned. There may, however, be tokens in the output place of some timers, e.g. mp_{2c} in Figure 6, which were never used for producing the output. This signifies that it was not yet time to fire those timers, i.e. the timer was fired prematurely. Before storing the new marking, the unused timer must therefore be "unfired" to reflect the fact that it was never used (Line 9 and Line 10), i.e. move the token from mp_{2c} back to mp_{2b}. The monitor is now ready again to be checked with respect to the timing behaviour of this marking in the next invocation of the assertion checker, according to the same procedure. After storing the new marking, the monitor has to restore the marking to the original situation (Line 5 and Line 12) before examining another enabled transition.

If the transition does not result in the desired output, the marking is restored (Line 12) and another enabled transition is examined. When all enabled transitions have been examined, the set *newmarkings* is returned (Line 13).

The algorithm will be illustrated with the sequence of inputs and outputs given in Equation (13) with respect to the monitor depicted in Figure 6 for the assertion in Equation (2). Port *p* is an in-port and *q* is an out-port. Initially, the set of current markings only consists of one marking, which is the initial marking of the monitor, formally denoted in Equation (14).

$$[p{=}\ 30, \neg p, \langle \text{delay:}20, p{=}\ 5\rangle, \neg p, \langle \text{delay:}2, p{=}\ 7\rangle, \neg p, \langle \text{delay:}5, q{=}\ 10\rangle, \neg q] \tag{13}$$

$$curmarkings\ =\ \{\{mp_1 \mapsto \langle 0, 0\rangle\}\} \tag{14}$$

The first transition puts a token in place *p* with the value 30. Since time has not elapsed, this operation is fine from the timing point of view. Putting tokens in an in-port is considered to be an input. The token is therefore just added to each possible current marking. The resulting set is shown in Equation (15).

$$curmarkings\ =\ \{\{mp_1 \mapsto \langle 0, 0\rangle, p \mapsto \langle 30, 0\rangle\}\} \tag{15}$$

In the next round, the token in *p* is consumed by the MUV. That is considered as an output, since it is an act by the MUV on its environment. According to the algorithm (Figure 8), the next step is to examine the enabled transitions and record the new possible markings. In this situation, four transitions are enabled in the monitor, mt_1, mt_2, mt_3 and mt_4. Firing mt_1 leads to a marking identical to the initial one, Equation (14), and mt_2 leads to the same marking but where a token has appeared in out-port *q*.

However, this is not the output marking stipulated by the MUV (no token in neither p nor q). For that reason, this marking is discarded. A similar argument holds for mt_3. Firing mt_4, on the other hand, leads to a marking where both mp_{2a} and mp_{2b} are marked and the output is the same as that of the MUV. Two markings are consequently valid considering the input and output observed so far. This is reflected in that *curmarkings* will contain both markings, as shown in Equation (16).

$$curmarkings = \{\{mp_1 \mapsto \langle 0, 0\rangle\}, \{mp_{2a} \mapsto \langle 0, 0\rangle, mp_{2b} \mapsto \langle 0, 0\rangle\}\} \tag{16}$$

The next input comes after 20 time units, when a new token appears in p, this time with the value 5. At this moment, time has elapsed since the previous transition firing. When the monitor is in the first marking, with a token in mp_1, only transitions mt_2 and mt_3 are enabled prior to giving the input to the monitor. Those transitions do not have an upper bound on their time delay interval. Therefore, in this marking, time can elapse without problem. However, in the second marking, with tokens in mp_{2a} and mp_{2b}, one transition is enabled, mt_6. Moreover, the upper time bound of that transition is 10 time units. Delaying for 20 time units will exceed this bound. As a conclusion, this marking is not valid and removed from the set of current markings. The input is then added to the remaining marking. The result is shown in Equation (17).

$$curmarkings = \{\{mp_1 \mapsto \langle 0, 0\rangle, p \mapsto \langle 5, 20\rangle\}\} \tag{17}$$

With no delay, the token in p is then consumed. As discussed previously this is considered to be an output. In this case, since the value of the token is 5, only three transitions are enabled, mt_2, mt_3 and mt_4. Transition mt_1, is disabled since its guard is not satisfied. Transitions mt_2 and mt_3 do not produce the same output as the MUV (consume the token in p), so they are ignored. Only mt_4 satisfies the output. The resulting set of markings is shown in Equation (18).

$$curmarkings = \{\{mp_{2a} \mapsto \langle 0, 20\rangle, mp_{2b} \mapsto \langle 0, 20\rangle\}\} \tag{18}$$

After 2 time units, another token arrives in p, this time with value 7. Advancing time by 2 time units is acceptable from the point of view of the monitor, since the only enabled transition, mt_6, has a higher upper bound, 10>2. It is not explicit in Figure 9, but it is now necessary to remember that 2 time units are already used from mt_6, leaving only 8 time units before it has to be fired. The input is added to each (only one in this case) set of current markings, as shown in Equation (19).

$$curmarkings = \{\{mp_{2a} \mapsto \langle 0, 20\rangle, mp_{2b} \mapsto \langle 0, 20\rangle, p \mapsto \langle 7, 22\rangle\}\} \tag{19}$$

Next, that new token disappears. Before examining the enabled transitions, all enabled timers must first tentatively be fired, leading to the markings in Equation (20).

$$curmarkings = \{\{mp_{2a} \mapsto \langle 0, 20\rangle, mp_{2c} \mapsto \langle 0, 22\rangle, \\ p \mapsto \langle 7, 22\rangle\}\} \tag{20}$$

Next, the token in p is consumed. Three transitions are enabled, mt_5, mt_7 and mt_8. However, only mt_5 satisfies the output. The resulting marking should consequently be stored. Firing transition mt_5 did not involve the timer (mt_6), so before storing the marking, the timer must be unfired, i.e. moving the token from mp_{2c} back to mp_{2b}. This was apparently not the right moment to fire the timer. Equation (21) shows the resulting marking.

$$curmarkings = \{\{mp_{2a} \mapsto \langle 0, 22\rangle, mp_{2b} \mapsto \langle 0, 20\rangle\}\} \tag{21}$$

After 5 time units, the MUV produces the output q with value 10. Again, the timer mt_6 is tentatively fired. Three transitions are now enabled, mt_5, mt_7 and mt_8, but only the latter two can produce a valid output. They lead to two different markings. The token in the output place of the timer, mp_{2c}, was consumed so no timer needs to be unfired. The result in shown in Equation (22).

$$curmarkings = \{\{mp_1 \mapsto \langle 10, 27\rangle, q \mapsto \langle 10, 27\rangle\}, \\ \{mp_{2a} \mapsto \langle 0, 27\rangle, mp_{2b} \mapsto \langle 10, 27\rangle, q \mapsto \langle 10, 27\rangle\}\} \tag{22}$$

The output, q, is then consumed by the environment of the MUV (stimulus generator). Removing a token from an out-port is considered as an input, for which reason it is removed from each marking in *curmarkings*. The remaining set of markings is shown in Equation (23).

$$curmarkings = \{\{mp_1 \mapsto \langle 10, 27\rangle\}, \\ \{mp_{2a} \mapsto \langle 10, 27\rangle, mp_{2b} \mapsto \langle 10, 27\rangle\}\} \tag{23}$$

The following example will demonstrate how an unsatisfied assertion is detected. Consider the sequence of inputs and outputs in Equation (24) and the assertion and monitor in Equation (2) and Figure 6 respectively.

$$[p = 5, \neg p, \langle \text{delay:}20, q = 10\rangle] \tag{24}$$

When the input p, with the value 5, and the output "consuming the token in p" have been processed, the set of current markings in the assertion checker has reached the situation in Equation (25).

$$curmarkings = \{\{mp_{2a} \mapsto \langle 0, 0\rangle, mp_{2b} \mapsto \langle 0, 0\rangle\}\} \tag{25}$$

After 20 time units a token in out-port q with value 10 is produced. First, time is elapsed in the monitor. One transition, mt_6, is enabled, and it has an upper time bound of 10 time units. The transition thus exceeds this bound, which makes the marking being discarded. The set of current markings is now empty, which signifies that the assertion is violated.

9 Coverage Enhancement

The previous sections have discussed issues related to the simulation phase (see Figure 1). The simulation phase ends when the stop criterion, which will be discussed in Section 10, is reached. After that, the validation algorithm enters the coverage enhancement phase, which tries to deliberately guide the simulation into an uncovered part of the state space, thereby boosting coverage. As indicated on Line 8 in Figure 1, a coverage enhancement plan has to be obtained. This plan describes step by step how to reach an uncovered, with respect to the particular coverage metrics used, part of the state space. This section describes the procedure to obtain the coverage enhancement plan. Obtaining this plan is the core issue in the coverage enhancement phase.

A model checker returns a counter-example when a property is proven unsatisfied. That is true for ACTL formulas. However, for properties with an existential path quantifier, the opposite holds. A witness is returned if the property is satisfied. A common name for both counter-examples and witnesses is diagnostic trace. For instance, when verifying the property **EF** φ, the model checker provides a trace (witness) which describes exactly step by step how to reach a marking where φ holds, starting from the initial marking. This observation is the centrepiece in the coverage enhancement procedure. The trace constitutes the coverage enhancement plan mentioned previously.

What φ represents depends on the particular coverage metrics used. In our case, the coverage metrics is a mix of assertion coverage and transition coverage as described in Section 5. The following two sections will go into the details of the peculiarities of enhancing both assertion and transition coverage respectively.

9.1 Enhancing Assertion Coverage

Each assertion has an associated activation sequence, as described in Section 6. During the simulation phase, the first markings in the sequence are removed as they are observed in the MUV. When the validation algorithm (Figure 1) reaches the coverage enhancement phase, the remaining activation sequences might therefore be partial. The first marking in the sequence with the least number of remaining markings is chosen as an objective, φ, for coverage enhancement.

Assume that no marking in the sequence corresponding to the property in Equation (11) has yet been observed, then the objective would be $p<20$, i.e. to find a sequence of transitions, such that when fired, would lead to a marking where $p<20$. The property given to the model checker would therefore be **EF** $p<20$. The model checker will then automatically provide the requested sequence of transitions in the diagnostic trace.

9.2 Enhancing Transition Coverage

Enhancing transition coverage is about finding a sequence of transitions leading to a marking where a previously unfired transition is enabled and fired. Having found a previously unfired transition, *t*, the property **EF** *fired*(*t*) is given to the model checker.

The model checker will then automatically provide a sequence of transitions, which, when fired, will lead to a marking where *t* is enabled and fired. In this way, transition coverage is improved.

The time that the model checker spends on finding a coverage enhancement plan depends heavily on which previously unfired transition is chosen for coverage enhancement. It is therefore worth the effort to find a transition which is "close" to the current marking, in the sense that the resulting enhancement plan is short, and hence also the model checking time. Definition 11 defines a measure of distance between a marking and a transition or place in PRES+ models. The measure can be used to heuristically find an appropriate transition that leads to a short trace. The measure, in principle, estimates the number of transition firings needed to fire the transition given the marking. It is chosen due to its fast computation.

Definition 11. Distance. Let *M* be a marking, *V* a set of values, **U** a universe containing all possible values which can occur in the model ($V \subseteq \mathbf{U}$), *t* a transition, *p* a place and c_1 and c_2 predefined constants. $P_t(i)$ denotes the *i*th input place of *t*. In Section 4.2 f_t and g_t were defined as the transition function and guard of transition *t* respectively. The function *dist*(*t*, *V*, *M*) is recursively defined as:

- If $\{ f_t(x_1, \ldots, x_n) \mid g_t(x_1, \ldots, x_n) \} \cap V = \varnothing$, then $dist(t, V, M) = c_1$

  Otherwise, $dist(t, V, M) =$

  $$\sum_{i=1}^{|\circ t|} dist(P_t(i), \{ v \in \mathbf{U} \mid g_t(x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_{|\circ t|}) \wedge$$

  $$f_t(x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_{|\circ t|}) \in V \wedge$$

  $$x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{|\circ t|} \in \mathbf{U} \}, M)$$

- If $M(p) \neq \varnothing \wedge M(p)_v \in V$, then $dist(p, V, M) = 0$

  If $M(p) \neq \varnothing \wedge M(p)_v \notin V$, then

  $$dist(p, V, M) = c_2 + \min_{t \in \circ p} \{ dist(t, V, M) \}$$

  If $M(p) = \varnothing$, then

  $$dist(p, V, M) = 1 + \min_{t \in \circ p} \{ dist(t, V, M) \}$$

V is an auxiliary parameter with the initial value $V = \mathbf{U}$. In the case of measuring the distance from a transition *t*, the set *V* contains all possible values which can be produced by the transition function f_t. Similarly, in the case of measuring the distance from a place *p*, *V* contains all possible values which a token in *p* can carry.

The distance between a transition *t* and a marking is defined in two different ways, depending on if there exist parameters to the function f_t, x_1, ..., x_n, which satisfy the guard g_t, such that the function can produce a value in the set *V*. If such parameters do

not exist, it means that the transition cannot produce the specified values. The distance is then considered to be infinite, which is reflected by the constant c_1. c_1 should be a number bigger than any finite distance in the model.

Otherwise, if at least one value in V can be produced by f_t, the distance of t is the same as the sum of all distances of its input places. The set V contains, in each invocation corresponding to input place p, the values which the function parameter associated to p may have in order for f_t to produce a value in V.

In the case of measuring the distance between a place p and a marking M, the result depends on whether there is a token in that place, and if so, the value in that token. If there is a token in p and the value of that token is in V, the distance is 0. In other cases, the search goes on in all of the incoming paths. For a token to appear in p, it is sufficient that only one input transition fires. The distance is defined with respect to the shortest (in terms of the distance) of them. This case is further divided into two cases: there is a token in p (but with a value not in V), or there is no token in p. In the latter case, 1 is added to the path to indicate that one more step has to be taken along the way from M to p. However, in the former case, a larger constant c_2 is added to the distance as penalty in order to capture the fact that the token in p first has to disappear before a token with an appropriate value can appear in p. The proposed distance heuristic does not estimate further the exact number of transition firings it takes for this to occur.

Figure 11 shows an example which will be used to illustrate the intuition behind the distance metrics. In the example, the distance between transition t_6 and the current marking (tokens in p_1 and p_2) will be measured. All transition functions are considered to be the identity function.

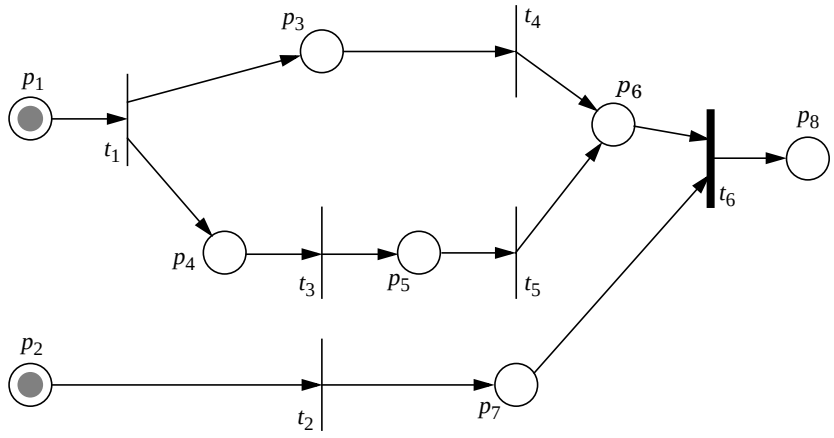

Figure 11. Example of computing distance

Transition t_6 has two input places p_6 and p_7. Consequently, in order to fire t_6 there must be tokens in both of these places. The distance of t_6 is therefore the sum of the distances of p_6 and p_7.

In order for a token to appear in p_7, only t_2 needs to be fired. The distance of p_7 is therefore 1 + $dist(t_2, \mathbf{U}, M)$. Transition t_2 is already enabled, so the distance of t_2 is 0 (because there is a token in its only input place). The distance of p_7 is hence 1.

At p_6, a token may appear from either t_4 or t_5. The distance of p_6 is therefore 1 plus the minimum distance of either transition.

The distance of t_5 is 2 (obtained in a similar way as in the case of p_7), while the distance of t_4 is 1. Therefore, $dist(p_6, \mathbf{U}, M) = 1 + \min\{dist(t_4, \mathbf{U}, M), dist(t_5, \mathbf{U}, M)\} = 2$.

Consequently, the distance of t_6 is $dist(t_6, \mathbf{U}, M) = dist(p_6, \mathbf{U}, M) + dist(p_7, \mathbf{U}, M) = 2 + 1 = 3$. Three transition firings are thus estimated to be needed in order to enable t_6.

Given this distance metrics, the uncovered transition with the lowest distance with respect to the current marking may be chosen as a target for coverage enhancement, since it results (heuristically) in the shortest enhancement plan, and it is obtained fast by the model checker.

This procedure can be taken one step further. Not only can the closest transition be chosen, but the smallest transition-marking pair. Among all visited markings and uncovered transitions, the pair with the smallest distance is chosen. When such a pair has been found, the model is reset to the particular marking and the coverage enhancement is performed with respect to that marking.

Although some time has to be spent on finding the transition-marking pair with the smallest distance, it is worth the effort since the time spent in the model checking can be reduced significantly. This is the alternative which we have implemented and used in the experiments. However, it is of great importance that the distance computation is as efficient as possible, since it is invoked many times when searching for a good transition-marking pair. In order to avoid long computation times, a maximum search depth can be introduced. When that depth is reached, a constant c_3 is returned, denoting that the distance is large.

9.3 Failing to Find a Coverage Enhancement Plan

It might happen that the model checking takes a long time. In such cases, a time-out interrupts this procedure leading to a situation where no coverage enhancement plan could be obtained. When this occurs, the rest of the coverage enhancement phase is skipped, and a new run of the simulation phase is started. The failed assertion or transition will not be targeted for coverage enhancement again.

10 Stop Criterion

Line 3 in Figure 1 states that the simulation phase ends when a certain stop criterion is reached. Section 3 briefly mentioned that the stop criterion holds when a certain number of transitions are fired without any improvement of the coverage. This number is called *simulation length*:

Definition 12. Simulation length. The simulation length is a parameter indicating the maximum number of consecutive transition firings during a simulation run without any improvement in coverage.

It can, however, be very difficult to statically determine a simulation length which minimises total validation time. In this section, a dynamic heuristic approach, where the simulation length is determined at run-time, is presented. Section 11 will demon-

strate that this heuristic yields a comparable coverage as the “optimal” simulation length with little penalty in time for the average case.

10.1 Static Stop Criterion

The diagram in Figure 12 depicts the relation between the simulation length and the total validation time. The graph shows the result of an example which has been validated several times with different values on the simulation length, with all simulation phases of one validation having the same value on the simulation length, σ. In other words, as soon as σ transitions had been consecutively fired without any increase in coverage, the simulation phase ended. Each diamond (marked “Sample”) in the figure corresponds to one such run. The example was validated 3 times per simulation length[1]. The average of the three results belonging to the same simulation length is also marked in the graph. The averages for each simulation length are connected by a line, in order to make the trend more visible[2].

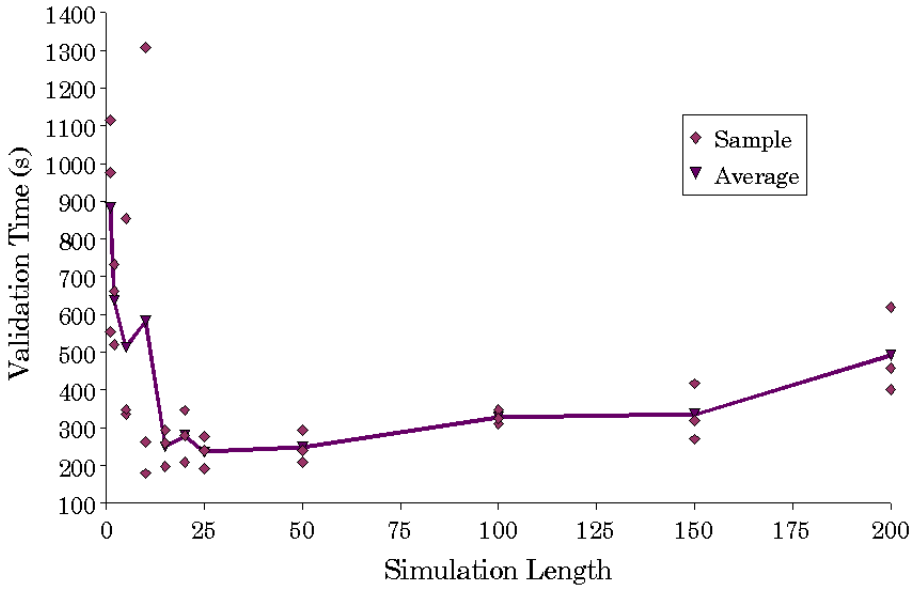

Figure 12. Relation between simulation length and validation time

For small simulation lengths, the validation time is quite high. This is due to the fact that not many transitions (or assertions) are covered in the simulation phase. Therefore, the model checker has to be invoked frequently to find enhancement plans for each of the uncovered transitions or assertions. This action is expensive. On the other hand, for big simulation lengths, a lot of time is spent in the simulation phase without any contribution to coverage. Between these two extremes, there is a point at which the least possible amount of time is wasted in the simulation phase and coverage enhancement phase respectively. For the example

1. Since there exists non-determinism both in the MUV and in the methodology, the verification times may differ even if the same simulation length is used.
2. All instances have been run until their actual finish according to Line 2 in Figure 1. This might, in general, lead to situations where the obtained coverage does not reach 100%. However, in this particular case, 100% coverage (according to Definition 8 in Section 5) is reached everywhere.

given in Figure 12, that point is somewhere between 15 and 30.

There are basically two factors that influence the location of this point: the size of the MUV (or actually its state space) and the sizes of the assertions (or actually the state space of their monitors). The bigger the size of the MUV is, the longer is the “optimal” simulation length, since more time will be needed for model checking in the coverage enhancement phase. Similarly, the bigger the sizes of the assertions are, the shorter is the “optimal” simulation length, since more time has to be spent in assertion checking in the simulation phase.

However, it is impossible to obtain the best simulation length before-hand without first validating the MUV multiple times. For this reason, a dynamic method which finds this value at run-time, while paying attention to coverage, is desired.

10.2 Dynamic Stop Criterion

As concluded in the previous section, the total validation time depends on the two factors: MUV size and assertion size. These factors influence model checking time and transition firing time respectively. The total validation time, consequently, is directly dependent on these factors. The following discussion will derive a function describing the total validation time in terms of these factors. This function will later be used to analytically determine the simulation length which most likely minimises validation time.

The total time the validation process spends in the coverage enhancement phase is the sum of the times of all model checking sessions. For a given simulation length, σ, and average model checking time (including the time to find a marking-transition pair with small distance), t_{ver}, the time spent in the coverage enhancement phase can be expressed as stated in Equation (26).

T is the set of transitions in the MUV and A is the set of assertions. $cov(\sigma)$ is a function expressing how big coverage would have been achieved at a certain simulation length σ, if the verification only is performed with the simulation phase, i.e. no coverage enhancement takes place. 1 - $cov(\sigma)$ thus denotes how big percentage of the state space has not been covered. It is assumed that this part has to be targeted in the coverage enhancement phase. Multiplying this number with the total number of transitions and assertions, $(1 - cov(\sigma)) \cdot |T \cup A|$, yields the number of transitions and assertions which are not yet covered by the simulation phase and need to be targeted by the coverage enhancement phase. Given that t_{ver} represents the average time spent in one coverage enhancement phase, the expression in Equation (26) approximates the total time spent in the coverage enhancement phase.

$$t_{enh}(\sigma) = (1 - cov(\sigma)) \cdot |T \cup A| \cdot t_{ver} \tag{26}$$

The time spent in the simulation phase depends mainly on the assertion checking time. The assertion checking procedure is invoked after each transition firing. The total time spent in the simulation phase is hence linear to the number of transitions fired.

The number of transitions fired can be approximated with the simulation length, in particular for large values. Equation (27) shows the corresponding expression, where t_{fir} is the average time it takes to fire a transition, including the subsequent assertion check.

$$t_{sim}(\sigma) \approx t_{fir} \cdot \sigma \tag{27}$$

The approximate total validation time is the sum of $t_{enh}(\sigma)$ and $t_{sim}(\sigma)$, as shown in Equation (28).

$$t_{tot}(\sigma) = t_{enh}(\sigma) + t_{sim}(\sigma) = (1 - cov(\sigma)) \cdot |T \cup A| \cdot t_{ver} + t_{fir} \cdot \sigma \tag{28}$$

The goal is to find a suitable simulation length σ, such that $t_{tot}(\sigma)$ is minimised. In Equation (28), there are three parameters whose values are unknown prior to validation, $cov(\sigma)$, t_{ver} and t_{fir}. The latter two can relatively straightforwardly be obtained by measuring the time it takes to run the model checker or fire and assertion check a transition respectively. Those parameters do not depend on the simulation length, but remain fairly constant throughout the entire process. Deviations will be captured in these average time measurements and exercise their relative impact in the computation of the next simulation length.

Before the first simulation phase, t_{ver} must be assigned an initial estimated value since it is too computationally expensive to invoke the model checker in order to obtain an authentic value. Not until after the first coverage enhancement phase, an authentic value has been obtained which can be used in the subsequent simulation phases. This initial value should depend on the size of the MUV.

Since it is relatively inexpensive to fire a transition and assertion check the result, t_{fir} does not need to be assigned an initial default value. After a few iterations in the simulation phase, an authentic value can quickly be obtained.

In order to determine the function $cov(\sigma)$, the following experiment is performed. First run the simulation phase with simulation length 1, and note down the obtained coverage in a diagram. Next, do the same with simulation lengths 2, 3, 4, etc. In practice, it is not necessary to run the simulation separately for each simulation length, but the coverage values for all simulation lengths can be obtained in one single run.

The experiment will show that the longer the simulation length is, the higher is the obtained coverage. For short simulation lengths, many uncovered transitions and assertions are encountered on each simulation length, resulting in a rapid rise in coverage. The longer the simulation length, the less additional uncovered transitions and assertions are encountered compared the previous simulation length. This function is, thus, exponential.

It is, however, practically infeasible to empirically derive this function by extensive simulation. By doing that, a large part of the system has already been verified and the use of knowing the coverage function, in order to minimise total validation time, is severely diminished.

Since we cannot know the exact shape of the *cov*(σ) function in advance, it has to be estimated. As the simulation phase progresses, more and more data about the coverage function can be collected, in the same way as the experiment described previously (Figure 12). The data collected for short simulation lengths obtained early during simulation, can be used to predict the coverage for longer simulation lengths.

Figure 13 shows a diagram in which the coverage obtained at each simulation length is marked (Sample) for a certain run. The graph only shows the results referring to one single simulation phase. Such diagrams follow in general an exponential curve of the form $cov(\sigma) = Ce^{D\sigma} + E$, where both C and D are negative. In order to obtain an estimation of the coverage function for instant simulation lengths longer than the one currently reached, these three parameters (C, D and E) have to be extracted from the points obtained from the simulation so far. This extraction is performed using the *Least Square Method*, under the constraint that $E \leq 1$. That method minimises the distance between the sample points (measured coverage) and the estimated curve.

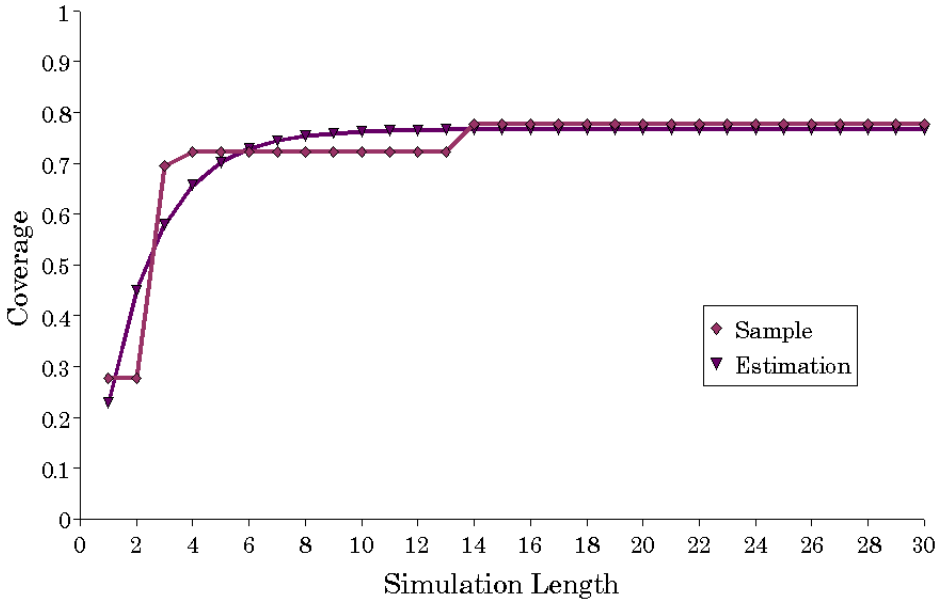

Figure 13. Relation between simulation length and coverage

Given the points in Figure 13, the least square method gives us that C = -0.9147, D = -0.53 and E = 0.7668. The resulting exponential curve is shown in the same figure (Estimation). Having obtained this function, Equation (28) can be rewritten as Equation (29).

$$t_{tot}(\sigma) = (1 - Ce^{D\sigma} - E) \cdot |T \cup A| \cdot t_{ver} + t_{fir} \cdot \sigma \qquad (29)$$

Provided the expression of total validation time in Equation (29), it is straightforward to find the simulation length corresponding to the shortest validation time using analytical methods. The resulting expression is presented in Equation (30).

$$\sigma = \frac{\ln \frac{t_{fir}}{C \cdot D \cdot |T \cup A| t_{ver}}}{D} \tag{30}$$

In the example of Figure 12 and Figure 13, t_{fir} = 0.1255s, t_{ver} = 33.36s and $|T \cup A|$ = 36. Using the formula in Equation (30) gives us that the optimal simulation length is 16, as computed in Equation (31).

$$\sigma = \frac{\ln \frac{0.1255}{(-0.9147) \cdot (-0.53) \cdot 36 \cdot 33.36}}{-0.53} \approx 16 \tag{31}$$

For each new simulation phase (Line 3 in Figure 1), a new simulation length is calculated according to Equation (30). The new value will be more accurate than the previous ones, as more data, on which the calculation is based, have been collected.

The only reason for not reaching 100% coverage with this simulation technique, is that obtaining a coverage enhancement plan took too long time and timed out. In such cases, the parameter t_{ver} will be large. Analysing Equation (30) gives that the larger t_{ver} is, the larger simulation length σ should be chosen (considering that both C and D are negative, and that the quotient inside the logarithm generally is less than 1). As a consequence, when more and more coverage enhancement attempts fail, the simulation phase becomes longer. Spending more time in the simulation phase results in a bigger collection of encountered markings (states), which increases the probability of finding a transition-marking pair with a small distance (Definition 11), and thereby influences coverage in a positive way.

11 Experimental Results

The proposed formal method-aided simulation technique has been evaluated on a variety of models and assertions. Each model was validated with both the static and dynamic stop criterion, as well as with pure model checking and pure simulation. The result of the comparison is presented in Table 1. The values given are the average values of several runs on the same model and assertions. The validations were performed on a Linux machine with an Intel Pentium 4 2.8GHz processor and 2GB of memory.

Table 1 Experimental results

		Time (s)		Time Diff.	Coverage (%)		Cov. Diff	Time (s)	Cov (%)
Ex.	**Trans.**	**Dyn.**	**Static**	**(%)**	**Dyn.**	**Static**	**(%)**	**MC**	**Sim**
1a	28	22.67	24.54	-7.62	100	100	0.00	OutMem	95
1b	28	51.75	38.30	35.12	100	100	0.00	OutMem	96
2a	35	42.40	39.74	6.69	100	100	0.00	OutMem	98
2b	35	62.55	60.78	2.91	100	100	0.00	OutMem	97
3a	42	55.38	58.77	-5.77	100	100	0.00	OutMem	97
3b	42	82.67	78.71	5.03	100	100	0.00	OutMem	94
4a	49	70.45	80.42	-12.40	100	100	0.00	OutMem	98
4b	49	101.64	105.16	-3.35	100	100	0.00	OutMem	97

Table 1 Experimental results

Ex.	Trans.	Time (s) Dyn.	Time (s) Static	Time Diff. (%)	Coverage (%) Dyn.	Coverage (%) Static	Cov. Diff (%)	Time (s) MC	Cov (%) Sim
5a	56	93.60	378.28	-75.26	100	99	1.01	OutMem	97
5b	56	143.36	120.73	18.74	100	100	0.00	OutMem	98
6a	63	137.14	289.89	-52.69	100	99	1.01	OutMem	99
6b	63	157.23	161.75	-2.79	100	100	0.00	OutMem	98
7a	70	298.81	151.43	97.33	99.5	100	-0.50	OutMem	100
7b	70	345.21	196.22	75.93	99.5	100	-0.50	OutMem	100
8	7	6.29	4.43	41.99	100	100	0.00	653.23	90
9a	14	261.98	399.91	-34.49	95	95	0.00	OutMem	100
9b	14	270.23	277.01	-2.45	95	95	0.00	OutMem	100
10a	21	627.27	550.78	13.89	95	94	1.06	OutMem	100
10b	21	891.39	821.83	8.46	89	90	-1.11	OutMem	100
11a	7	7.57	4.66	62.45	100	100	0.00	609.63	90
11b	7	16.41	10.49	56.43	100	100	0.00	379.20	86
12a	14	253.19	240.65	5.21	98	95	3.16	OutMem	100
12b	14	265.08	388.45	-31.76	93	95	-2.04	OutMem	100
13	30	15.27	10.42	46.55	100	100	0.00	OutMem	100
14	75	119.37	93.06	28.27	100	100	0.00	OutMem	98
15	150	564.54	504.37	11.93	100	100	0.00	OutMem	99.5
16	225	1768.35	1604.84	10.19	100	100	0.00	OutMem	99
17a	31	1043.97	935.68	11.57	98	99	-1.01	OutMem	84
17b	31	599.19	417.30	43.59	95	100	-5.00	OutMem	91
18a	36	216.41	157.01	37.83	100	100	0.00	OutMem	86
18b	36	279.46	250.10	11.74	100	100	0.00	6037.53	86
19a	8	13.12	10.21	28.50	100	100	0.00	OutMem	83
19b	8	330.47	316.21	4.51	100	100	0.00	N/A	92

The complexity of the models is given in terms of number of transitions. Although this number is generally not enough to characterise the complexity of a model, it still provides a hint about the size of the model and is used due to the lack of a more accurate, but still concise, metrics. The number only includes the transitions located inside the MUV itself, not the ones modelling the stimulus generators. So despite the relatively small number of transitions in the examples, the statespace might become quite large, also due to the parallel nature of PRES+ models and the vast range of possible values tokens may take.

Examples 1 through 16 consist of artificial models, and examples 17 through 19 model a control unit for a mobile telephone, traffic light controller and a multiplier respectively. The models are often verified for two different properties, why they appear twice in the table. The model of the mobile telephone is moreover described in more detail in Appendix B.

Generating the artificial models must be done under simultaneous consideration of the assertion to be checked. The model must be generated in such a way that the assertion will be satisfied in it. It is therefore difficult to apply purely random methods. The models therefore consist of multiple predefined pieces of Petri-nets connected to each other, forming a model that satisfies a given assertion by construction. This provides an efficient way to generate test examples of different sizes. The assertions in the examples are typical and frequently used reachability properties.

It should be emphasised that the simulation length used for the static stop criterion was obtained by empirically evaluating several different values, finally choosing the one giving the shortest validation time for comparison. The performance given by this method can consequently not be achieved in practice, and only serves as a reference.

As can be seen, in most cases using the dynamic stop criterion results in validation times close to those for the static stop criterion. There exist however cases where there is a big difference. This situation occurs if one method did not reach as high coverage as the other. Thus, the coverage enhancement phase must have failed due to a time-out, resulting in longer total time.

It can be deduced from the figures that, in average, the dynamic approach is 15% slower than using the static stop criterion (if that one could be used in practice, which is not the case). However, in 30% of the cases, the dynamic approach was actually faster. This situation may occur, since choosing the simulation length for the static stop criterion is not a very accurate process. It could happen that the dynamic stop criterion finds a simulation length closer to the actual optimum. The loss in coverage is, on the other hand, very small. In average, coverage is 0.12% lower using the dynamic approach.

Although the dynamic approach performs slightly worse on both aspects, it should be remembered that, in practice, it is impossible to reach the values listed in the table with the static approach. As mentioned previously, the values were obtained by trying several alternatives for the static simulation length, thus validating the system multiple times. It cannot be known in advance which simulation length results in the shortest validation time.

For comparison, all models and all assertions have, in addition, been validated with only model checking and only simulation. In the model checking case, the model checker ran out of memory (2GB) in most cases before a solution was found. However, in the cases where a solution was found, the model checker spent one or two orders of magnitude longer time than the proposed mixed approach. In one example (19b) the model checker did not support the verified property, for which reason no result is given for that example.

As for pure simulation, it was let running for as long time as was used in the case of the dynamic stop criterion (Table 1). The total coverage obtained by the simulation process after this time, was in most cases less than or equal to our proposed mixed approach. This is in particular true for the non-random models (17-19).

The general conclusion that can be drawn regarding pure model checking and simulation is that in most cases the proposed approach outperforms both. Although pure model checking guarantees 100% coverage, it takes significantly longer time to obtain a result, if a result can be obtained at all due to memory limitations. On the other hand, pure simulation cannot reach as high coverage as the proposed approach given the same amount of time.

12 Conclusions

This paper has presented a simulation technique into which formal methods (model checking) have been injected to boost coverage. The approach has been developed on a Petri-net based design representation called PRES+, and on the two orthogonal coverage metrics transition and assertion coverage. However, the underlying mechanisms are also applicable to any transition-based design representation and to any coverage metrics.

The simulation algorithm consists of two distinct and successive phases: simulation and coverage enhancement. During the simulation phase, transitions are repeatedly selected from the model under verification and fired. The results from the firings are validated by an assertion checker. When it can be concluded that pure simulation will not lead to any increase in coverage, it stops and control is passed on to the coverage enhancement phase. In that phase, a so far uncovered part of the state space is identified and targeted by applying model checking. As a result from the model checking, a sequence of transitions leading to the previously identified uncovered state is obtained, thereby improving coverage. After that, a new round of the simulation phase is initiated starting from the previously uncovered state identified by the enhancement phase.

Moreover, a technique to automatically generate stimulus generators and assertion checkers from ACTL formulas is used. These generators and checkers are modelled in the same PRES+ representation as the MUV and, in the case of stimulus generators, they are connected to the MUV so that they appear as one entity from the point of view of transition selection. The transition selection mechanism is, moreover, biased in order to target scenarios described by the assertions, thereby enhancing assertion coverage already in the simulation phase.

In order to minimise total validation time, the model checker is dynamically invoked. The invocation time is continuously estimated based on historical values on certain parameters influencing performance, with the objective of minimising total validation time. This estimation is performed once after each coverage enhancement phase, yielding more and more accurate estimations at each iteration.

The approach has been demonstrated by experiments to be effective. It outperforms pure simulation on models where pure model checking is too memory and time consuming. Verification times have, in addition, shown to be close to optimum.

Appendix A

The methodology relies on the fact that a (T)CTL formula can be captured by a PRES+ model: for the stimulus generator and for the assertion checker. This appendix very briefly outlines a procedure which performs this transformation on ACTL properties. The resulting PRES+ model captures all behaviours that satisfy the formula. The detailed procedure is presented in [27].

The fundamental idea behind this algorithm [27] is to find the maximum set of behaviours or states which do not violate the

particular ACTL formula at hand [28]. A PRES+ model corresponding to this set of behaviours is then constructed. Let us outline the algorithm considering the example ACTL formula in Equation (32). It states that if there is a token in port *p*, the value of that token must be less than 2.

$$\mathbf{AG}\ (p \rightarrow p < 2) \tag{32}$$

It is found that there are two states which satisfy the example formula. They are described by Equation (33) and Equation (34) respectively. Either there is no token in *p* and the property continues to hold (Equation (33)), or there is a token in *p* with a value less than 2 and the property continues to hold (Equation (34)).

$$\neg p \wedge \mathbf{AXAG}\ (p \rightarrow p < 2) \tag{33}$$

$$p < 2 \wedge \mathbf{AXAG}\ (p \rightarrow p < 2) \tag{34}$$

In this simple example, there is only one way in which the property may continue to hold (**AXAG** ($p \rightarrow p < 2$)). Except for the port *p*, the resulting model only has one place, corresponding to this future behaviour. In general, there is one place per such future behaviour. Cases with multiple possibilities of future behaviour arise, for instance, when several temporal operators occur in the formula.

Figure 14 shows the resulting model. It contains one place corresponding to the port *p* and another place (s_1) corresponding to **AXAG** ($p \rightarrow p < 2$).

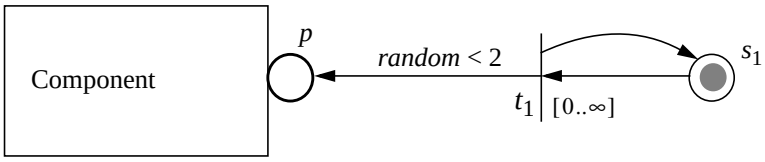

Figure 14. The PRES+ model corresponding to the example formula in Equation (32)

For each place (possible future behaviour), transitions must be added for each possible state. There are two such states in the example, Equation (33) and Equation (34). For Equation (33), no transition needs to be added since it says that *p* should *not* have any token. The only action to take would be to stay in the same place. For Equation (34), a token must be put in place *p* with a random value less than 2, as performed by transition t_1. A token must also be put in the place corresponding to the continuation, in this case the same place. There is no time requirement on this action. Consequently, the transition may fire at any time, as reflected by time delay interval [0..∞]. An initial token is finally added to the place.

Appendix B

This appendix illustrates the mobile telephone controller, one of the real-life examples, that was verified in the experimental results (Section 11).

The mobile telephone controller keeps track of what is happening in the overall system and acts accordingly. Figure 15 shows a model of the component.

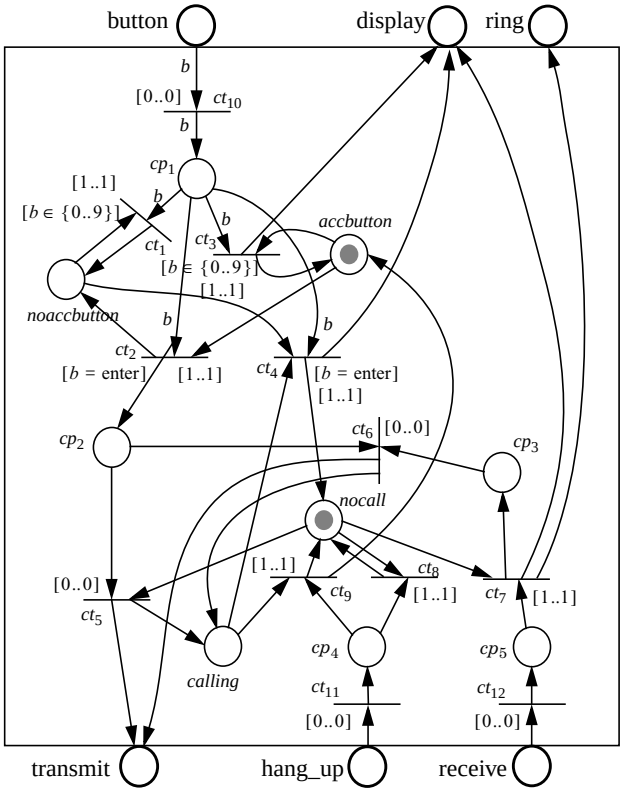

Figure 15. Model of the Controller component

Places $accbutton$ and $noaccbutton$ are marked when the controller is able or is not able to process button data respectively. The data is simply discarded if it is not immediately accepted. Transitions ct_1 to ct_4 take care of this functionality. The transitions have guards so that different actions can be taken depending on which button was pressed. This model only makes a difference between if a number was pressed, $b \in \{0..9\}$, or if "enter" was pressed, $b = \text{enter}$. When dialling a number, signals (tokens) are also sent in order to update the display. Having pressed "enter" the telephone number is sent to the transmitter.

Places $calling$ and $nocall$ record whether a phone call is taking place or not. Transition ct_5 therefore updates these places when a phone call is to be made. Transition ct_7 takes care of incoming phone calls and ct_8 and ct_9 handle the end of a call.

Assumptions were moreover added to the ports of the components to provide input. These assumptions were given as ACTL formulas and converted into PRES+ using the technique presented in Appendix A. An illustrative example of such an assumption formula is given in Equation (35).

$$\mathbf{AG}(button \rightarrow button \in \{0, \ldots, 9, \text{enter}\} \wedge \mathbf{AXAG}_{<5} \neg button) \tag{35}$$

The formula states that the valid range of values of tokens in port $button$ is the numbers 0 to 9 and the enter key. The time interval between two consecutive tokens in that port must be longer than 5 time units.

References

[1] Haase, J., “Design Methodology for IP Providers”, Proc. Design and Test in Europe, Munich, Germany, Mar 1999, pp. 728-732.

[2] Gajski, D., A C.-H. Wu, V. Chaiyakul, S. Mori, T. Nukiyama, P. Bricaud, “Essential Issues for IP Reuse”, Proc. Asia and South Pacific Design Automation Conference, Yokohama, Japan, Jan 2000, pp. 37-42.

[3] Bryant, R.E., “Graph-Based Algorithms for Boolean Function Manipulation”, Trans. Computers, Vol. C-35, No 8, pp. 677-691, 1986

[4] Piziali, A., “Functional Verification Coverage Measurement and Analysis”, Kluwer Academic Publishers, 2004

[5] Wang, F., A. Mok, E. A. Emerson, “Symbolic model-checking for distributed real-time systems”, Lecture Notes in Computer Science, Vol. 670, 1993

[6] Daws, C., S. Yovine, “Reducing the number of clock variables of timed automata”, Proc. IEEE Real-Time Systems Symposium, Washington DC, USA, Dec 1996, pp. 73-81

[7] Balarin, F., “Approximate reachability analysis of timed automata”, Proc. IEEE Real-Time Systems Symposium, Washington DC, USA, Dec 1996, pp. 52-61

[8] Seger, C.-J.H., R.E. Bryant, “Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories”, in Formal Methods in Systems Design, pp. 6:147-189, 1995

[9] Tasiran, S., Y. Yu, B. Batson, “Linking Simulation with Formal Verification at a Higher Level”, IEEE Design & Test of Computers, Vol. 21:6, 2004

[10] Synopsys whitepaper, “Hybrid RTL Formal Verification Ensures Early Detection of Corner-Case Bugs”, 2003

[11] Shyam, S., V. Bertacco, “Distance-Guided Hybrid Verification with GUIDO”, Proc. Design and Test in Europe, Munich, Germany, Mar 2006, pp. 1211-1216

[12] de Paula, F., A.J. Hu, “An Effective Guidance Strategy for Abstraction-guided Simulation”, Proc. Design Automation Conference, San Diego, USA, Jun 2007, pp. 63-68

[13] Lu, Y., L. Weimin, “A Semi-formal Verification Methodology”, Proc. International Conference on ASIC, Shanghai, China, Oct 2001, pp. 33-37

[14] Abts, D., S. Scott, D.J. Lilja, “So Many States, So Little Time: Verifying Memory Coherence in the Cray X1”, Proc. International Parallel and Distributed Processing Symposium, Nice, France, Apr 2003

[15] Krohm, F., A. Kuehlmann, A. Mets, “The Use of Random Simulation in Formal Verification”, Proc. International Conference on Computer Design: VLSI in Computer and Processors, Austin, USA, Oct 1996, pp. 371-376

[16] Gupta, A., A.E. Casavant, P. Ashar, X.G. Liu, A. Mukaiyama, K. Wakabayashi, “Property-Specific Testbench Generation for Guided Simulation”, Proc. Asia South Pacific Design Automation Conference, Bangalore, India, Jan 2002, pp. 524-531

[17] Nanshi, K., F. Somenzi, “Guiding Simulation with Increasingly Refined Abstract Traces”, Proc. Design Automation Conference, San

Fransisco, USA, July 2006, pp. 737-742

[18] Hessel, A., K.G. Larsen, B. Nielsen, P. Pettersson, A. Skou, “Time-Optimal Real-Time Test Case Generation using UPPAAL”, Proc. International Workshop on Formal Approaches to Testing of Software, Montreal, Canada, Oct 2003, pp. 118-135.

[19] Fey, G., R. Drechsler, “Improving Simulation-based Verification by Means of Formal Methods”, Proc. Asia South Pacific Design Automation Conference, Yokohama, Japan, Jan 2004, pp. 640-643

[20] Clarke, E.M., E.A. Emerson, A.P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”, Trans. Programming Languages and Systems, pp. 8(2):244- 263, 1986

[21] Alur, R., C. Courcoubetis, D.L. Dill, “Model Checking for Real-Time Systems”, Proc. Symposium on Logic in Computer Science, Philadelphia, USA, 1990, pp. 414-425

[22] Cortés, L.A., P. Eles, Z. Peng, “Verification of Embedded Systems using a Petri Net based Representation”, Proc. International Symposium on System Synthesis, Madrid, Spain, Sep 2000, pp. 149-155

[23] Alur, R., D.L. Dill, “A theory of timed automata”, Theoretical Computer Science, pp. 126:183-235, 1994

[24] Karlsson, D., P. Eles, Z. Peng, “Formal Verification of SystemC Designs Using a Petri-net based Representation”, Proc. Design and Test in Europe, Munich, Germany, Mar 2006, pp. 1228-1233.

[25] UPPAAL homepage: http://www.uppaal.com/

[26] Oliveira, M.T., Hu, A.J., “High-Level Specification and Automatic Generation of IP Interface Monitors”, Proc. Design Automation Conference, New Orleans, USA, Jun 2002, pp. 129-134

[27] Karlsson, D., “Verification of Component-based Embedded System Designs”, Ph.D. Thesis, Linköping Studies in Science and Technology Dissertation No. 1017, Linköpings universitet, 2006, `http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-7473`

[28] Grumberg, O., D.E. Long, “Model Checking and Modular Verification”, Transactions on Programming Languages and Systems, Vol 16 No 3, pp. 843-871, 1994