Grade/CPN: A Tool and Temporal Logic for Testing Colored Petri Net Models in Teaching

Michael Westergaard[1,2,⋆], Dirk Fahland[1], and Christian Stahl[1]

[1] Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
{m.westergaard,d.fahland,c.stahl}@tue.nl
[2] National Research University Higher School of Economics,
Moscow, 101000, Russia

Abstract. Grading dozens of Petri net models manually is a tedious and error-prone task. In this paper, we present Grade/CPN, a tool *supporting the grading of Colored Petri nets modeled in CPN Tools*. The tool is extensible, configurable, and can check static and dynamic properties. It automatically handles tedious tasks like checking that good modeling practise is adhered to, and supports tasks that are difficult to automate, such as checking model legibility. We propose and support the *Britney Temporal Logic* which can be used to guide the simulator and to check temporal properties. We provide our experiences with using the tool in a course with 100 participants.

1 Introduction

Colored Petri nets (CPNs) [1] is a formalism useful for modeling a broad range of real-life systems, including complex network protocols [1] and business information systems [2]. It is thus natural to use CPNs or other Petri net formalisms when teaching such subjects. As modeling can only really be learned by doing, hands-on experience is a must. Larger classes can comprise more than one hundred students, and manually checking models created by students is time consuming and error-prone. This is particularly unpleasant because much of the effort is spent on checking trivial things, including whether good modeling standards are adhered to and whether formal requirements to the model are satisfied. In this paper, we aim at *supporting the grading of many models implementing the same specification* by providing with Grade/CPN an *extensible tool* for automatic assessment of such routine properties, allowing teachers to focus on more complicated tasks.

The support required for grading assignments is similar to what is needed for testing or model checking, as we need to check a model against some formal requirements. The models we deal with in our case study have infinite state spaces, so here we focus on the testing perspective, as a model may not be suitable for

⋆ The study was implemented in the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE) in 2013.

M. Koutny et al. (Eds.): ToPNoC VIII, LNCS 8100, pp. 180–202, 2013.

model checking due to having a large or even unbounded state space. Thus, parts of the work described here is also applicable to general testing of CPN models, but we present it here in the context in which it was developed. The significant difference to classical testing is that for grading *a possibly large set of different models* is to be checked against *the same specification* in a uniform way.

CPN Tools [3] is a tool for editing, simulating and analysing CPN models. It supports the user during the construction of the model due to incremental syntax checking, which gives immediate feedback about errors, and allows modelers to experiment with incomplete and even only partially correct models. This is a useful feature for inexperienced users and makes CPN Tools suitable in teaching. Furthermore, the Windows version of CPN Tools is downloaded more than 5,000 times a year, indicating that it is broadly used. The broad usage also means that CPN Tools has reached a fairly stable state, which reduces unnecessary frustrations during modeling. Finally, CPN Tools has extensive online help and video tutorials, which means it is easy for students to get started. For these reasons, we think that CPN Tools is a good choice of a tool for teaching.

There are as many ways of using models as there are teachers, so it is important that the requirements for the model can be described easily. This means that the grading tool must be *configurable*, allowing individual teachers to customize what is checked and how adhering to or deviating from each requirement is awarded or punished. In addition, it must be easily possible to *extend* the tool with new requirements. Thus our tool must have a plug-in like architecture allowing new requirements to be added with minimal effort. At the same time, we do not desire a heavy-weight framework with a steep learning curve just to add a simple custom requirement. Of course, such a tool should come with a set of reasonable built-in plug-ins, so it is useful for many scenarios without requiring any programming.

To illustrate our motivation for developing such a tool, assume we want students to model a (simplified) delivery service using CPN Tools. The idea is to model that customers order products from a shop, and the shop uses a delivery service to deliver ordered products to the customers. To this end, we would provide students with a base model as in Fig. 1. The CPN in Fig. 1 models the behavior of the customer and the shop and provides the interface between customer and delivery service (Reject, Offer, Accept, and Delivery) and the interface between shop and delivery service (Shipment, Return, and Notification). A customer can choose a product from the catalog and place an order via place Order. The shop prepares the ordered product for shipment and sends the resulting packet to the delivery service via Shipment.

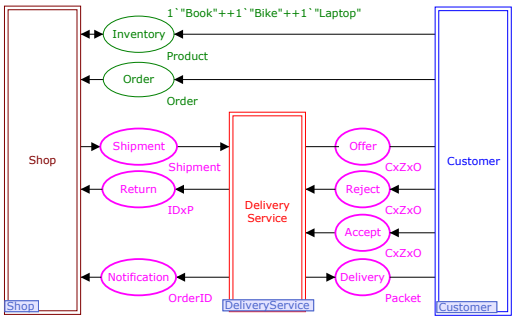


Fig. 1. Base model of a delivery service

The delivery service shall in all tasks try to deliver packets to the respective customers via place Offer. If a customer is not at home, a token is placed on place Reject; otherwise, a token is produced on place Accept and, finally, the delivery service hands over the packet to the customer via place Delivery. Place Return is used to send a packet back to the shop in case the packet could not be delivered. In addition, the delivery service informs the shop via place Notification that a packet has been successfully delivered. The pages Shop and Customer are given but the DeliveryService is empty and intended to be modeled by the student.

When students are given such a base model, they are asked to model the missing part(s) or to change or improve the given model. These changes must adhere to certain constraints. In our example, we would need to be able to check that the given *environment has not been changed* (as the environment constitutes a contract with the external world) and that the *model satisfies the given requirements*, which often means that behavioral properties need to be checked. Our focus on the first version of our tool has therefore been on making it easy to check these requirements.

We have also implemented checks that ensure good modeling practice, including *respecting data hiding* (i.e., student solutions are not allowed to connect to nodes of the environment other than the interface places) and *proper termination* (i.e., ensuring that tokens are not erroneously left behind), and simple *static analysis* (e.g., ensuring that communication channels are used in the correct direction, i.e., no messages are produced on an input channel).

As we cannot check all properties mechanically—for example, whether the model is readable and understandable—we have implemented functionality supporting doing this manually. This includes generating a *view of the model* in which the student-designed parts are highlighted and the given parts from Fig. 1 are dimmed. This allows teachers to focus on the new parts without having to distinguish these parts manually.

We have earlier encountered problems with students copying solutions from one another. We would also like to detect this, so we have *checks that at least make it harder to cheat*. This includes providing each student with a unique copy of the base model from Fig. 1 with a cryptographic signature including the student ID embedded. This makes it impossible for two students to use the same base model as starting point (indicating that one got a copy from the other).

Finally, we want a *report* summarizing all findings; the report should be useful for both teachers, who should be able to grade the model based on the report only, without having to manually open the model in CPN Tools except in special cases, and for students, who should be pointed to flaws in the model, using error traces when applicable. To sum up, we need a tool that

1. Works with CPN Tools models.
2. Provides easy configuration.
3. Is easily extensible.
4. Contains a reasonable base set of capabilities, including:
   (a) Detect changes to a given environment,
   (b) Check dynamic properties using simulation.

   (c) Check good modeling practise, including data hiding, proper termination, and provide simple static analysis.
5. Supports the manual part of the grading process.
6. Detects attempts to defraud.
7. Provides a report that pin-points problems, aids the teacher in grading, and allows students to understand problems.

We have chosen to implement our tool as a vanilla Java application. The language is chosen due to its popularity and platform-independence. We have chosen not to rely on a framework for handling plug-ins, as these frameworks often demand significant overhead due to providing features we do not need (e.g., we do not need dynamic configuration of plug-ins). We have used the library Access/CPN [4] as it provides an easy way to load CPN models and programmatically interact with the simulator.

We continue with the outline of the architecture of our tool and introduce some simple plug-ins checking basic properties in Sect. 2. In Sect. 3, we introduce a temporal logic which is powerful enough to describe most dynamic requirements while still being easy to use. In Sect. 4, we provide details on automatic attempt to improve coverage and relate our work to automatic testing. In Sect. 5, we provide implementation details and we sum up our experiences using our tool in semi-automatically assessing assignments from close to 100 students. Finally, we discuss related work, conclude the paper, and provide directions for future work.

A preliminary version of this work has been published in [5]. Compared with [5], we have extended our syntax to handle a global quantifier and variables, and provide a simpler semantics with subtle errors fixed (see Sect. 3). Moreover, the details on coverage and the comparison with automatic testing (see Sect. 4) are new results. We have also implemented some of the future work of the previous paper, including a version of the tool allowing students to get feedback before final grading (see Sect 4), and we report how that has improved the grades of students (see Sect. 5).

2 Architecture

In this section, we outline the architecture of Grade/CPN. We first give the overall architecture and explain how this solves requirements 1, 2, 3, and 7 from the introduction. Then, we provide the details of some of the built-in plug-ins, focusing on requirements 4(a), 4(c), and 6. Requirement 4(b) is handled in detail in the next section, and requirement 5 is handled partly in this section and partly when we report our experiences in Sect. 5.

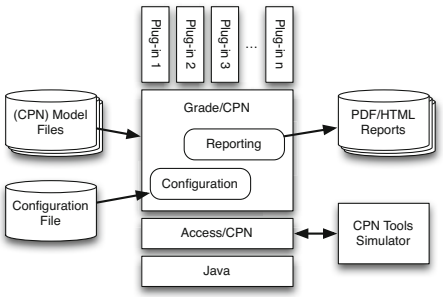


Fig. 2. Overall architecture and environment of Grade/CPN

2.1 Overall Architecture

Figure 2 shows the overall architecture of Grade/CPN. We see that we build on top of Java and Access/CPN [4]. Access/CPN is a Java library making it possible to interact directly with the CPN Tools Simulator, including loading models and translating them to an object structure we can use for static analysis, and send to the simulator process also used by CPN Tools to perform syntax check and simulation of models. Grade/CPN comprises two important components, one for Configuration and one for Reporting, as well as an interface to several Plug-ins. The Configuration component is responsible for loading a configuration file and using it to instantiate and configure the appropriate plug-ins. Each plug-in returns messages useful for the Reporting component, which use this information to generate an on-screen status view showing the overall correctness of the checked models and for generating an individual report for each student. The report can be generated as either an HTML file suitable for reading in a Web-browser or a PDF file suitable for printing or archival.

The central interface of Grade/CPN is PlugIn, shown in Listing 1 (ll. 1–5). Each plug-in must implement this interface. The configure method is a factory method to instantiate the plug-in, and takes how many points should be awarded if the plug-in succeeds and a configuration string. The format of the configuration string is defined by the plug-in, but will typically be a name identifying the plug-in and a list of named parameters. If the plug-in can be instantiated with a given configuration string, it returns a new configured instance and otherwise it returns null. This allows us to create an abstract factory for instantiating plug-ins from a string. Furthermore, a plug-in has a method grade, which is given a student ID, a base model (base), the student solution (model), and a connection to the simulator. The plug-in can use this information to arrive at its conclusion and return a Message, which comprises how many points are awarded and a descriptive message with the reason for the grade.

Reporting. The Reporting component of Fig. 2 is responsible for emitting a report based on the result of the PlugIns. All interfaces pertaining to reporting is shown in Listing 1 (ll. 7–17). The main class is Report (ll. 7–10), which is instantiated for each student ID and contains a set of pairs of PlugIns and Messages (produced by the grade method).

Point range	Points	Reason
-100.00 - 0.00	0.00	The interface has not been modified incorrectly.
-5.00 - 0.00	0.00	Declarations were preserved and new ones were added (that is ok).
-5.00 - 0.00	-5.00	generated_Task1b-solution.cpn is not a substring of Task1b-solution.cpn
-5.00 - 0.00	-5.00	Did not match with 0 < 65
0.00 - 0.03	0.03	30 Random Orders was executed successfully 10 times
0.00 - 0.03	0.03	Packet to Depot after Reject was executed successfully 10 times

Fig. 3. Report overview

A Message (ll. 11–13) ties together a number of awarded points, a descriptive message and a list of Details providing in-depth reasoning leading to the outcome. Each Detail (ll. 14–17) consists of a descriptive header and either a list of textual details or a single graphical component, which is rendered as an image in the resulting report. For each student a report overview is generated (see Fig. 3 for an example) and supplementary details are added in separate sections.

Listing 1. Plug-in interface and central components

```
1  public interface PlugIn {
2    public PlugIn configure(double maxPoints, String configuration);
3    public Message grade(StudentID id, PetriNet base, PetriNet model,
4                         HighLevelSimulator simulator);
5  }

7  public class Report {
8    public Report(StudentID sid) { ... }
9    void addReport(PlugIn plugin, Message result) { ... }
10 }
11 public class Message {
12   public Message(double points, String message, Detail... details) { ... }
13 }
14 public class Detail {
15   public Detail(String header, String... details) { ... }
16   public Detail(String header, JComponent component) { ... }
17 }

19 public class Tester {
20   public Tester(TestSuite suite, List<StudentID> ids, PetriNet base) { ... }
21   public List<Report> test() { ... }
22 }
23 public abstract class TestSuite {
24   public TestSuite(PlugIn matcher) { ... }
25   public abstract List<PlugIn> getPlugIns();
26 }
27 public class ConfigurationTestSuite extends TestSuite {
28   public ConfigurationTestSuite(File configurationFile) { ... }
29 }
```

Configuration. The Configuration component of Fig. 2 is shown in Listing 1 (ll. 19–29). The main class is a Tester (ll. 19–22), which given a TestSuite, a list of student IDs, and a base model can perform a test (l. 21) and yields a Report for each student. A TestSuite (ll. 23–26) has a distinguished matcher, which is a PlugIn mapping models to student IDs by yielding a high score for a model and student ID pair if the model is created by the student with the given ID and a low score otherwise. A TestSuite can also return a list of PlugIns for the main grading process. One implementation of a TestSuite, the ConfigurationTestSuite (ll. 27–29), is instantiated using a configurationFile which along with an abstract PlugIn factory is used to instantiate the correct PlugIns according to the configuration.

An example configuration file is shown in Listing 2. The file comprises two sections, matcher (ll. 1–2) and test (ll. 4–15), setting up the matcher and the actual tests graded, respectively. The intuition is that each line corresponds to a plug-in; a line starting with a + (ignoring white space) is considered part of the preceding line. Each line starts with a number indicating how many points are awarded for successful execution. If the number is negative, successful execution yields 0 points but a failure yields a punishment. This is followed by a colon and a configuration option recognized by the plug-in and optionally a list of named parameters. For example, in line 5 we see that the plug-in identified by declaration-preservation is instantiated with one named parameter. If the test fails, it yields a punishment of 5 points and if it succeeds, it yields 0 points. Lines 13–14 are merged (as line 14 starts with +). In the following we go into more detail with this example.

2.2 Simple Plug-ins for Interface Preservation

In Listing 2, we use two plug-ins to ensure that the interface to the environment and the environment itself are not modified. The declaration-preservation plug-in (l. 5) makes sure that no declaration in the provided model is removed or

Listing 2. Example configuration file

```
1  [matcher]
2  -5: signature,threshold=65

4  [tests]
5  -5: declaration-preservation
6  -100: interface-preservation,addpages=true,initmark=true,subset=deliveryservice
7  -5: matchfilename
8  0.033: btl,repeats=2,name="Accept 10 Orders",test=
9    + (10 * (--> Order) -> (@(!Order))) &
10   + (10 * (--> Receive) -> (@(!Receive))) &
11   + (@(!Reject)) &
12   + [(--> Handle_Return) => false]
13 0.033: btl,repeats=2,name="Only two cars of capacity 1",test=
14   + [@(|Reject| + |Offer| + |Accept| < 3)]
```

changed. This ensures that it is impossible to change the type of the interface by redefining color sets. If declarations are removed or changed, this is reported as an error and if new declarations are added, they are added to the report so it is easy to see what was added without having to directly compare the student model with the base model.

The interface-preservation (l. 6) plug-in makes sure that students do not change the given net structure, but only add new structure. In our example from Fig. 1, students are only allowed to add new net structure, but not to modify the given environment. Here, we are given four parameters. The addpages parameter is set to true, which means that students are allowed to add new pages. The initmark option is set to true, which means that students are not allowed to change the initial marking of the model. Finally, the subset parameter contains a list of pages students are allowed to add structure to. Any page not in this list is not allowed to be changed at all. Here, we specify that the students are allowed to alter the DeliveryService page from Fig. 1. Any added page is listed in the report as is any modified page. If the change is illegal, the error is listed (i.e., if a node of the interface is removed or altered, this is highlighted), and if the model contains no errors, the entire environment is dimmed so only the student solution is highlighted.

2.3 Fraud Prevention

We have two plug-ins for matching a model to a student ID. In Listing 2, we use both to award points. We see in line 8 that we instantiate the matchfilename plug-in. This plug-in simply checks if the student ID is a substring of the filename (and punishes if it is not). This is fine for honest students; unfortunately, we have in earlier years encountered students copying models from one another. To catch that, we instead use the more elaborate signature plug-in as matcher (l. 2).

The signature matcher exploits that all elements of a CPN Tools model have a unique identifier. This is necessary, e.g., to represent that an arc is connected to a specific place and transition. While these identifiers must be unique in the file and match for nodes and arcs, the actual contents of the identifiers have no semantics. We have developed a simple signer application which, given a base model, modifies the identifiers in a predictable way. By using a cryptographic random number generator, we can generate a sequence of pseudo-random

numbers using the student ID and a secret passphrase as seed. The idea is that if we know the passphrase and the student ID, we can regenerate the sequence, but using just the sequence (and optionally the student ID), it is not possible to reverse-engineer the passphrase. Now, using the generated sequence of numbers as identifiers of model elements in the file containing the environment, we create a unique signature in the base file for each student.

The signature plug-in can check this signature. It queries for each student ID and student model whether the two match. It regenerates the sequence of random numbers for the student ID and the provided passphrase, and check that the identifiers are present in the file. If they are, the model is considered a match and otherwise not. The plug-in takes a parameter threshold which indicates how many identifiers must be present in the model. As the signing is a one-way process, students are forced to use the appropriate base model and cannot just hand in the same model (even after making cosmetic changes). The teacher only needs to remember the password as the signature key is generated automatically from the password and student ID.

2.4 Model-Checking

Grade/CPN embeds the ASAP model-checker [6], allowing us to check all properties supported by that tool as long as the state-space is finite, including LTL properties using a wide range of reduction techniques. The models we encounter in our case study do not have finite state-spaces, so we have not been focused on this part. The extensible nature of Grade/CPN makes it easy to add this functionality externally, and as a proof-of-concept we have implemented a simple dead-lock checker.

3 Britney Temporal Logic

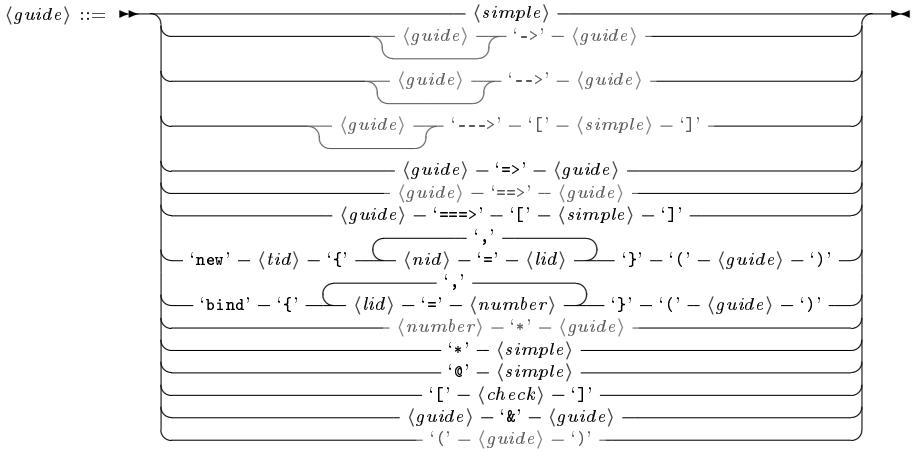
An important requirement to our tool is to check dynamic properties, requirement 4(b) from the introduction. In the example in Fig. 1, we are for example interested in the behavior when a customer accepts packets ten times in a row and how many packets can be outstanding at any time. As CPN models tend to have huge or even infinite state spaces, we cannot verify such properties in general and especially not for models generated by students who have less experience with modeling. Therefore, we check such properties by guiding the model; that is, we apply a testing-based approach rather than exhaustive state-space exploration, yielding a sound but not necessarily complete checking mechanism.

Guiding the model requires to specify which transition the model should execute. Testing whether some property holds in a state of the model requires a specification of this property. To this end, we introduce the *Britney Temporal Logic* (BTL). This logic is similar to linear-time logic (LTL) [7] but, in addition to checking properties, also allows guiding the model and to specify constraints that should hold in a state. We adopt a syntax more similar to common descriptions of Petri net firing sequences rather than cryptic abbreviations or symbols

to make it easier for practitioners to adopt the logic. The choice for an LTL-like logic reflects our wish to have existential counterexamples that can be represented by a simple firing sequence. Other kinds of counterexamples are difficult to find using simulation only and also difficult to present to the user. In the following, we define the syntax of BTL formulae and then their semantics based on Kripke structures [8], and structural operational semantics (SOS) [9] to capture invariant properties and simple rewrite rules to capture the temporal aspects.

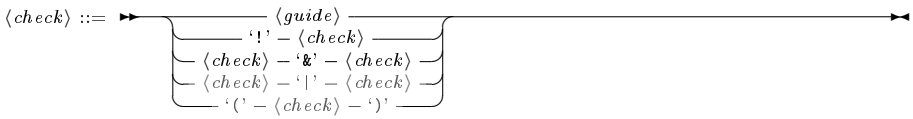
3.1 Syntax

A BTL formula is a ⟨*guide*⟩. A guide describes how simulation should be performed; that is, it guides the model to a desired state. The atomic propositions of a guide are described using ⟨*simple*⟩, which is an expression without temporal operators but otherwise allowing full propositional logic on transitions and place invariants. The temporal operators are six arrows emulating the arrows typically used to describe transition steps. Thus *a*`->`*b* means that first *a* must hold and subsequently *b* must hold. For example, *a* and *b* can represent transitions, meaning that for the formula to hold, the corresponding transitions are executed one after the other. We lift this operator to *a*`-->`*b* meaning that *a* must hold and sometime afterward *b* must hold. Finally, *a*`--->`[*b*] means that *a* must hold and when the simulation stops *b* must hold. The brackets indicate that *b* is not used for guiding the simulation anymore (it has terminated after all). We can omit *a*, which is an abbreviation for *true*. For each arrow, we also add a double arrow version indicating that *if a* holds, then *b* holds at the appropriate time.

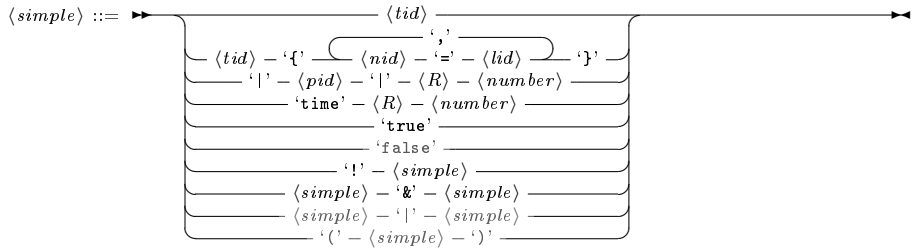


We use operator `new` to define that the firing of a transition initializes one or more variables, and we use operator `bind` to initialize one or more variables with constant values. We also allow bounded and unbounded repetition using a star syntax. In contrary to a regular Kleene star, we put it in front as it improves readability for western readers. Using operator @, a guide can specify an invariant property that should hold in all states. A guide can also include ⟨*check*⟩s, which are not used to guide the model but only to test assertions.

They are therefore allowed to contain disjunctions and negations and general boolean expressions. Finally, a guide can also be the conjunction of two guides.



In addition to the syntax for guiding, we also allow simple boolean expressions. These are mostly for testing state properties, such as counting the tokens on a place or testing values of the global clock. Attribute *tid* and *pid* in the grammar thereby refer to a transition label and place label, respectively, and *nid* is the name of a CPN variable and *lid* is the name of a local BTL variable. In this definition, constants are only bound to numbers for sake of simplicity, however the extension to arbitrary CPN literals is straight forward. In the syntax, symbol $\langle R \rangle$ is any of the comparison operators $<, >, \leq, \geq, =$. For example, Line 12 in Listing 2 tests that Handle Return is never executed (but does not enforce it like the guides). The formula in line 14 checks that at any point during execution, the three places Reject, Offer, and Accept never contain three or more tokens in total.



Our syntax includes a lot of conveniences. We already mentioned that avoiding the precondition for the single arrows is a convenience for a precondition of *true*. Furthermore, all single arrows can be defined from the double arrows by forcing the precondition. The eventuality defined by a==>b can be defined in terms of the unbounded repetition and the next operator, and bounded repetition is just a syntactical convenience. Let G, G_1, G_2 be guides, C, C_1, C_2 be checks, and S, S_1, S_2 be simple boolean expressions. In the syntax, we have grayed out all syntactic sugar for which we do not need to explicitly define the semantics.

$$
\begin{aligned}
\texttt{->}G &\equiv \mathit{true}\texttt{->}G \\
\texttt{-->}G &\equiv \mathit{true}\texttt{-->}G \\
\texttt{--->}[C] &\equiv \mathit{true}\texttt{--->}[C] \\
G_1\texttt{->}G_2 &\equiv G_1\&(G_1\texttt{=>}G_2) \\
G_1\texttt{-->}G_2 &\equiv G_1\&(G_1\texttt{==>}G_2) \\
G_1\texttt{--->}[S] &\equiv G_1\&(G_1\texttt{===>}[S]) \\
G_1\texttt{==>}G_2 &\equiv G_1\texttt{->}(*\mathit{true}\texttt{->}G_2)
\end{aligned}
\qquad
\begin{aligned}
(G) &\equiv G \\
C_1|C_2 &\equiv !(!C_1\&!C_2) \\
(C) &\equiv C
\end{aligned}
\qquad
\begin{aligned}
\mathit{false} &\equiv !\mathit{true} \\
S_1|S_2 &\equiv !(!S_1\&!S_2) \\
(S) &\equiv S
\end{aligned}
$$

$$
n * G \equiv \begin{cases} G\texttt{->}(n-1) * G & \text{if } n \geq 1 \\ \mathit{true} & \text{otherwise.} \end{cases}
$$

3.2 Semantics

The semantics of BTL is similar to a standard finite trace semantics for LTL like the one defined in [10]. Intuitively, `=>` corresponds to "next", `==>` corresponds to "eventually", @ corresponds to "globally" (in a restricted form), and a formula $[\langle check\rangle]$`==>`$\langle guide\rangle$ is similar to "until" (in a restricted form). Yet, BTL significantly differs from LTL due to the dual nature of guides (which steer the simulation) and checks (which have to hold).

We interpret BTL formulae over a Kripke structure $K = (Q, \delta, q_0, \Sigma, \lambda)$, where Q is a set of states, $q_0 \in Q$ is the initial state, Σ is a set of transition labels, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and function $\lambda : Q \to 2^{AP}$ maps each state $q \in Q$ to a set of atomic propositions that hold in q. As usual, AP denotes the set of all atomic propositions. In our syntax we have some CPN-specific atomic propositions dealing with places and time, but it obvious that these could be replaced to suit any formalism generating a Kripke structure.

The semantics of a BTL formula is defined over the traces of a Kripke structure K along with an environment, E, which is a function mapping names to values. Normally for a model M, we consider the transition relation $\to_M$ relating two states q_0, q_1 and a transition. What exactly the state and transitions are depends on the concrete formalism. In the case of CPNs, the states are markings and the transitions are pairs consisting of a transition and all variables surrounding it, a *binding element*. We denote by $BE \subseteq T \times 2^{Bindings}$ the set of all possible binding elements for a model, and we write $q_0 \xrightarrow{t\{n_1=v_1,\ldots,n_j=v_j\}}_M q_1$ to denote the model can execute transition t with the binding of variables $n_1 = v_1, \ldots, n_j = v_j$ from state q_0, leading to state q_1. We say that the binding element $t\{n_1 = v_1, \ldots, n_j = v_j\}$ is *enabled* and denote by $name(t\{n_1 = v_1, \ldots, n_j = v_j\}) = t$ the transition name of a binding element.

We first consider how to guide the simulation. This is done by defining a set of allowed transitions for each guide. For simulation, only enabled transitions that are in this set are considered. This in particular means that if the set of allowed transitions is empty, the simulation is considered finished (and not with an error unless the formula is not satisfied). In other words, when considering the truth value of a formula according to a model (without a trace), we only consider the truth value along all traced adhering to the guides. We define the set *guide* over a set of possible binding elements inductively as follows, where S, S_1, S_2 are of type $\langle simple\rangle$, C, C_1, C_2 are of type $\langle check\rangle$, and G, G_1, G_2 are of type $\langle guide\rangle$:

$$
\begin{aligned}
guide(tid, q, E) &= \{be \in BE \mid name(be) = tid\} \\
guide(tid\{n_1 = l_1, \ldots, n_j = l_j\}, q, E) &= \{tid\{n_1 = E(l_1), \ldots, n_j = E(l_j)\}\} \\
guide(|pid|\ R\ i, q, E) &= guide(\text{time}\ R\ i, q, E) = BE \\
guide(\text{true}, q, E) &= BE \\
guide(!S, q, E) &= BE \setminus guide(S, q, E) \\
guide(S_1 \& S_2, q, E) &= guide(S_1, q, E) \cap guide(S_2, q, E) \\
guide(!C, q, E) &= BE \setminus guide(C, E) \\
guide(C_1 \& C_2, q, E) &= guide(C_1, q, E) \cap guide(C_2, q, E)
\end{aligned}
$$

$$
\begin{aligned}
guide(G_1 \texttt{=>} G_2, q, E) &= guide(G \texttt{===>}[S], q, E) = BE \\
guide(\text{new } tid\{n_1 = l_1, \ldots, n_j = l_j\}(G), q, E) &= \{tid\{n_1 = v_1, \ldots, n_j = v_j\} \in BE \mid \\
& \quad tid\{n_1 = v_1, \ldots, n_j = v_j\} \in guide(G, q, E[l_1 \mapsto v_1, \ldots, l_j \mapsto v_j])\} \\
guide(\text{bind } \{l_1 = v_1, \ldots, l_j = v_j\}(G), q, E) &= guide(G, q, E[l_1 \mapsto v_1, \ldots, l_j \mapsto v_j]) \\
guide(*S, q, E) &= guide([C], q, E) = BE \\
guide(@S, q, E) &= guide(S, q, E) \\
guide(G_1 \wedge G_2, q, E) &= guide(G_1, q, E) \cap guide(G_2, q, E)
\end{aligned}
$$

We allow concrete steps if they are needed to satisfy a formula or forbid a step if it would violate it, and otherwise allow anything when we do not care about the outcome.

Next, we define the semantics of $\langle simple \rangle$ over traces $q_0 \xrightarrow{be_1}_M q_1 \xrightarrow{be_2}_M \cdots \xrightarrow{be_k}_M q_k$ of enabled binding elements as follows. Most operators are straightforward with (1) consuming transitions and binding elements for non-empty traces, (2) defining state predicates, and (3) defining propositional connectives.

$$
\frac{k \geq 1,\ be_1 \in guide(tid, q_0, E)}{(q_0 \xrightarrow{be_1}_M \cdots q_k), E \models tid}
\qquad
\frac{k \geq 1,\ be_1 \in guide(tid\{n_1 = v_1, \ldots, n_j = v_j\}, q_0, E)}{(q_0 \xrightarrow{be_1}_M \cdots q_k), E \models tid\{n_1 = v_1, \ldots, n_j = v_j\}} \tag{1}
$$

$$
\frac{q_0 \models |pid|\ R\ i}{(q_0 \xrightarrow{be_1}_M \cdots q_k), E \models |pid|\ R\ i}
\qquad
\frac{q_0 \models time\ R\ i}{(q_0 \xrightarrow{be_1}_M \cdots q_k), E \models time\ R\ i} \tag{2}
$$

$$
\frac{}{(q_0 \xrightarrow{be_1}_M \cdots q_k), E \models true}
\qquad
\frac{(q_0 \xrightarrow{be_1}_M \cdots q_k), E \not\models S}{(q_0 \xrightarrow{be_1}_M \cdots q_k), E \models !S}
$$
$$
\frac{(q_0 \xrightarrow{be_1}_M \cdots q_k), E \models S_1 \wedge (q_0 \xrightarrow{be_1}_M \cdots q_k), E \models S_2}{(q_0 \xrightarrow{be_1}_M \cdots q_k), E \models S_1 \wedge S_2} \tag{3}
$$

The $\langle check \rangle$ is a simple syntactical extension of $\langle guide \rangle$ and treated with them. The operators on $\langle guide \rangle$ are LTL-like. As for $\langle simple \rangle$, we define the syntax over traces $q_0 \xrightarrow{be_1}_M q_1 \xrightarrow{be_2}_M \cdots \xrightarrow{be_k}_M q_k$ of enabled binding elements. Instead of defining the truth value, we need to define a rewrite of a formula to capture the temporal aspects as well as the guiding aspects. We define the *progress* function inductively on the structure of the union of $\langle guide \rangle$ and $\langle check \rangle$, execution trace, and an environment E. We notice that this includes *true* and *false*. A $\langle simple \rangle$ can always be evaluated in the current state or step according to rules (1)-(3).

$$
progress(S, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = \begin{cases} true & \text{if } (q_0 \xrightarrow{be_1}_M \cdots q_k, E) \models S \\ false & \text{otherwise} \end{cases} \tag{4}
$$

A $\langle guide \rangle$ or a $\langle check \rangle$ may evaluate to *true* or *false* in the current state or step, in which case we return this value. If not, the $\langle guide \rangle$ or $\langle check \rangle$ is rewritten to the formula that has to hold in the next step. Rule (5) shows the rewriting for the conditional next step construct, where G_2 has to hold in the next step if G_1

holds in this step, while the entire formula has to hold in the next step if nothing can be said about G_1 in this step. By (6), conditional "finally" is only evaluated at the end of the trace.

$$progress(G_1 \texttt{=>} G_2, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = \begin{cases} G_2 & \text{if } progress(G_1, q_0 \xrightarrow{be_1}_M q_1, E) = true \\ true & \text{if } progress(G_1, q_0 \xrightarrow{be_1}_M q_1, E) = false \\ (progress(G_1, q_0 \xrightarrow{be_1}_M \cdots q_k, E) \texttt{=>} G_2) & \text{otherwise} \end{cases} \tag{5}$$

$$progress(G \texttt{===>}[S], q_0 \xrightarrow{be_1}_M \cdots q_k, E) = \begin{cases} (q_0, E) \models S & \text{if } G = true, k = 0 \\ true & \text{if } G \neq true, k = 0 \\ (progress(G, q_0 \xrightarrow{b_1}_M \cdots q_k, E)) \texttt{===>}[S] & \text{otherwise} \end{cases} \tag{6}$$

Rule (7) replaces the dynamic binding of BTL variables by "new" with the static binding when the concrete values are known and the transition is allowed; the static binding recursively extends the environment for the subformulas (8).

$$progress(\texttt{new } name\{n_1 = l_1, \ldots, n_i = l_i\}(G), q_0 \xrightarrow{be_1}_M \cdots q_k, E) = \begin{cases} \psi & \text{if } be_1 = name\{n_1 = v_1, \ldots, n_i = v_i\} \\ false & \text{otherwise} \end{cases} \tag{7}$$

where $\psi = progress(\texttt{bind}\{l_1 = v_1, \ldots, l_i = v_i\}(G), q_0 \xrightarrow{be_1}_M \cdots q_k, E)$.

$$progress(\texttt{bind}\{l_1 = v_1, \ldots, l_i = v_i\}(G), q_0 \xrightarrow{be_1}_M \cdots q_k, E) = \begin{cases} true & \text{if } \psi = true \\ \texttt{bind}\{l_1 = v_1, \ldots, l_i = v_i\}(\psi) & \text{otherwise} \end{cases} \tag{8}$$

where $\psi = progress(G, q_0 \xrightarrow{be_1}_M \cdots q_k, E[l_1 \mapsto v_1, \ldots, l_i \mapsto v_i])$.

The "@S" defines a global invariant S that has to hold in each step of the trace until its end (10), the "$*S$" permits the simple S to hold on a prefix of the trace 9.

$$progress(*S, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = \begin{cases} *S & \text{if } k > 0, (q_0 \xrightarrow{be_1}_M q_1), E \models S \\ true & \text{otherwise} \end{cases} \tag{9}$$

$$progress(@S, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = \begin{cases} @S & \text{if } k > 0, (q_0 \xrightarrow{be_1}_M q_1), E \models S \\ true & \text{if } k = 0 \\ false & \text{otherwise} \end{cases} \tag{10}$$

Checks do not restrict the step: they are simply evaluated, or, if they cannot be evaluated, are rewritten according to the current step. We preserve syntactic categories of checks in rules (11) and (12) accordingly.

$$progress([C], q_0 \xrightarrow{be_1}_M \cdots q_k, E) = \begin{cases} true & \text{if } progress(C, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = true \\ false & \text{if } progress(C, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = false \\ [progress(C, q_0 \xrightarrow{be_1}_M \cdots q_k, E)] & \text{otherwise} \end{cases} \tag{11}$$

$$progress(!C, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = \begin{cases} false & \text{if } progress(C, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = true \\ true & \text{if } progress(C, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = false \\ [!progress(C, q_0 \xrightarrow{be_1}_M \cdots q_k, E)] & \text{otherwise} \end{cases} \quad (12)$$

A satisfied conjunction is rewritten to *true* as in rule (13); this rule equally applies to conjunctions over $\langle check \rangle$s.

$$progress(G_1 \wedge G_2, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = \begin{cases} progress(G_1, q_0 \xrightarrow{be_1}_M \cdots q_k, E) & \text{if } progress(G_2, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = true \\ progress(G_2, q_0 \xrightarrow{be_1}_M \cdots q_k, E) & \text{if } progress(G_1, q_0 \xrightarrow{be_1}_M \cdots q_k, E) = true \\ progress(G_1, q_0 \xrightarrow{be_1}_M \cdots q_k, E) \wedge progress(G_2, q_0 \xrightarrow{be_1}_M \cdots q_k, E) & \text{otherwise} \end{cases} \quad (13)$$

The *progress* function determines how to progress the computation for each step. Sometimes the computation cannot progress, however. This can be either because there are no more enabled transitions (the trace is empty) or the guard does not permit progressing. In this case, we need to check that the remaining rewritten formula can terminate, i.e., if it accepts the empty trace. We then lift the computation over traces from individual steps to entire traces. We define an *evaluate* function evaluating the truth value of a formula f over a trace $q_0 \xrightarrow{be_1}_M q_1 \xrightarrow{be_2}_M \cdots \xrightarrow{be_k}_M q_k$ of enabled transitions as follows. The function returns one of three values, *true* meaning the formula holds for the trace, $false$ meaning it does not hold, and *unguided* meaning the trace does not follow the guiding function.

$$evaluate(f, q_0 \xrightarrow{be_1} \cdots q_k) = \begin{cases} true & \text{if } progress(f, q_0, \emptyset) = true \\ false & \text{if } k = 0, progress(f, q_0, \emptyset) \neq true \\ unguided & \text{if } be_1 \notin guide(q_0, f, \emptyset) \\ evaluate(progress(f, q_0 \xrightarrow{be_1} \cdots q_k),\ q_1 \xrightarrow{be_2} \cdots q_k) & \text{otherwise} \end{cases} \quad (14)$$

Finally, we say that given a model M and a formula f, M satisfies the formula f, written $M \models f$ if all traces either satisfy the formula or are unguided, formally:

Definition 1 (Satisfaction of BTL). *Given a (CPN) model M and a BTL formula f, we say that M **satisfies** f, written $M \models f$ iff*

$$\forall q_0 \xrightarrow{be_1} \cdots q_k \in M : evaluate(f, q_0 \xrightarrow{be_1} \cdots q_k) \neq false$$

4 Coverage and Choices

Section 3 introduced syntax and semantics of BTL, which allows to specify intended and forbidden behavior of the system. In this section, we discuss how to *test* whether a system model, given as CPN model, satisfies the specification. We first discuss requirements for testing and our approach, including how to get high coverage and how these ideas can be used to handle choices in the form of disjunctions.

4.1 Testing BTL Formulas

Similar to formal verification, testing aims at finding errors in the system model – that is, finding runs which violate the given specification. In contrast to formal verification, testing is not exhaustive: Only a fraction of the system's possible behavior is investigated for whether it violates the specification.

A naive testing algorithm is to randomly walk through the state space of the system model, until the property tested for is satisfied or violated. This is repeated several times. If a run violating the specification is found, it is proof that the system violates the specification. In case no violating run is found, there is no proof that the system is error free. However, one can compute the probability by which a specification holds based on the explored behavior in relation to the complete behavior.

Figure 4 shows a technical example. For evaluating the BTL formula $A\text{-->}C$, only the *guided traces* of the execution tree are relevant as unguided traces have no impact on the satisfaction of a formula. We do not assign all traces the same probability, as simulation locally decides on each step regardless of any previous and future steps. In the first step, the system is guided to do an A, so all traces leaving the initial state with an action different from A are unguided and not considered. This means the traces have probability 1 of starting with an A. The remaining guided tree is shown in Fig. 4(top right). Edges correspond choices in the model and the probabilities show the probability of a random trace having the given prefix. We only consider completed traces, though we can sometimes make a decision prior to exploring full traces. We see that after executing A we have a choice between C and D, so each prefix amounts to half of the probability. The trace ACD satisfies the formula (we can already see it will after executing just the prefix AC). By just testing the trace prefix AC, we know that the entire system satisfies $A\text{-->}C$ with a probability of $\frac{1}{2}$, because the probability to see traces with this prefix is $\frac{1}{2}$. By exploring more alternatives, we see more traces and, in case all explored traces satisfy the formula, increase the probability that the formula holds, i.e., when also exploring ADC, which also has probability $1 \cdot \frac{1}{2} \cdot 1 = \frac{1}{2}$, we get that the system satisfies $A\text{-->}C$ with probability 1.

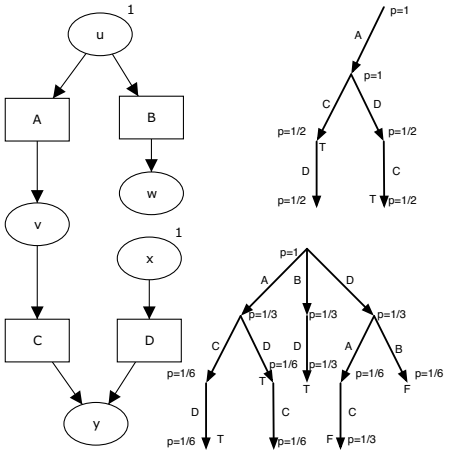


Fig. 4. Example model N, guided execution tree for $A\text{-->}C$, and guided execution tree for $!C\text{-->}D$

A different situation occurs for the formula $!C\text{-->}D$ which has the guided tree of Fig. 4(bottom right). The guide $!C$ does not prune any (enabled) behavior. If the test explores the trace DB or DAC, it finds a counterexample for the

formula proving it was violated. By non-exhaustive exploration in testing, we could also end up with just exploring the traces ACD (which has probability $\frac{1}{3} \cdot \frac{1}{2} = \frac{1}{6}$) and BD (which has probability $\frac{1}{3} \cdot 1 = \frac{1}{3}$), and in this case would have a total probability of $\frac{1}{6} + \frac{1}{3} = \frac{1}{2}$ that the formula holds. Once exploring a violating trace, such as DB, that probability drops to 0. Exploring the entire execution tree yields either probability 0 or 1 that the formula holds.

In general, we talk about three different percentages: the coverage, which is the weighted sum of all explored traces; the probability a formula holds, which is the weighted sum of all explored traces if they all satisfy the formula or zero otherwise; and the probability that a random trace satisfies the formula. When we have found no counter-examples these are the same, and obviously the larger the probability that a random trace satisfies the formula, the more difficult it is to find a counter-example. The example shows the main challenge in testing: to explore that fraction of traces that yield a counter-example, or if so such exists, yields a high probability that the formula holds (which increases confidence in the test result).

4.2 Heuristics for Higher Confidence in Test Results

In testing, confidence put into a test result is typically measured in terms of *coverage criteria* [11]. Various coverage criteria have been proposed such as *state coverage* (i.e., the fraction of place that has been marked at least once), *transition coverage* (i.e., the fraction of transitions that occurred at least once, regardless of binding element), or *coverage of all paths* (to a certain length). Coverage criteria are in some sense interchangeable, as one can simulate coverage w.r.t. one criterion by coverage w.r.t. another one [12].

Path Coverage. To improve the naive testing algorithm of repeatedly walking through the system state space in a random way, we leverage two coverage criteria to increase confidence that a CPN model satisfies a given BTL formula: transition coverage and path coverage. Complete coverage for paths is infeasible in the presence of loops or unbounded non-determinism, but covering paths up to a certain length is feasible. To increase path coverage, we essentially explore the guided execution tree of the CPN model by greedily choosing a branch that has the largest probability of falsifying a formula. In the simplest case with no information about the model, this is the one with the largest difference between the probability of a random trace having the prefix represented by the branch and the probability of the formula holding in that subtree. If we consider the example for $!C\text{-->}D$ in Fig. 4, assuming we have explored ACD, starting from an empty trace we would pick either B or D as they both have probability $\frac{1}{3}$ of happening and known probability 0 of the formula holding, whereas the subtree starting with A also has probability $\frac{1}{3}$ of happening and probability $\frac{1}{3}$ of holding.

Transition Coverage. Generating a test for complete coverage of all transitions is an undecidable problem in a CPN model, as for each transition, one would have to find a coloured firing sequence that enables this transition. For this reason, we apply heuristics when exploring the tree of guided executions. We maintain

a queue of all transitions of the CPN model. When deciding on the next step in a run, we pick the first enabled transition from the queue and after firing move it to the end of the queue. This way, we increase the chance of firing transitions that were not considered yet. Binding elements are not part of the queue and we also prefer branches giving higher coverage by using the previous heuristic.

More Advanced Criteria. We have assumed that the variables have no impact on the enabled traces. This is of course a simplification, and we could also consider trying to evaluate the guards to drive the model to different states, e.g., using abstraction. We could also use transition invariants (of the uncolored underlying model) to identify loops that are less likely to be interesting, or partially order the transitions according to pre and post places to try and drive a notion of progress in the model.

4.3 Disjunctions

We have avoided adding disjunction to our guides. This is primarily done because adding disjunctions can be very expensive. For example, an expression like $(A\texttt{-->}B)|(B\texttt{-->}D)$ must make a choice when used for guiding if both A and B can be executed. If, in the example in Fig. 4, A is chosen, a D and a C are encountered, and the execution terminated, we cannot conclude that the formula does not hold, as the second part of the disjunction was ignored. We therefore have to back-track and try again to ensure there really is an error, making handling disjunctions as difficult as model-checking.

Furthermore, the semantics of disjunction is not completely obvious as we make truth of formulae relative to the guide. In the formula $((A\texttt{-->}C)|(B\texttt{-->}D))$ $\texttt{-->}D$, must the system be able to respond to both $A\texttt{-->}C$ and $B\texttt{-->}D$ with a D or is it enough that the system responds with a D for one of the environment interpretations? As the truth is relative to the guide, either interpretation becomes unclear; normally we would make the guide of a disjunction the union of the guides for the two elements, but this makes the guide a larger set, which may yield strange results. For example, a system may respond to being guided by $A\texttt{-->}C$ with a D like in Fig. 4 (thus intuitively satisfying the system), but not respond to $B\texttt{-->}D$ with a D. If the system allows this behavior, this would mean that the disjunction is not true, even though one of its sides is; the disjunction inherits similarities to a conjunction (both sides must be satisfiable for the disjunction to be true). This interpretation is counter-intuitive (and contradicts the behavior of disjunction of simple formulae, e.g., $A|B$). The only way to get around that is to change the guide to instead return sets of sets of transitions, one for each branch of a disjunction.

If we split the guide to handle disjunctions, we need to check each set of guides. Each set would partition according to the left side, right side, and intersection of each disjunction all the way through the structure of the guide, causing the number of sets needed to explore to grow exponentially in the number of disjunctions.

Instead of dealing with this, which theoretically is manageable and nice, we have decided that disjunction in guides unnecessarily complicates the semantics

and complexity of checking. We can handle multiple environment behaviors by instead checking a formula for each individual environment behavior, and we can already test disjunctions in the simple boolean checks.

5 Practical Experience

In this section, we briefly present our implementation of BTL and Grade/CPN, and practical experiences of using both in a course.

5.1 Implementation

Our implementation of BTL uses simple formula rewriting according to the semantics. Our implementation implements the *guide* set for filtering enabled transitions, pick and execute one that is in the *guide* set and in the set of enabled transitions. We then rewrite the formula according to the previous rules. For efficiency, we have expanded some of the syntactical equivalences, most importantly the future temporal operator (a==>b). When no more transitions are in the intersection, we check if the rewritten formula is satisfied for the empty trace.

We evaluate formulae using a four-valued logic similar to [13]. The idea is that we have two versions of both true and false: The value is definite and can never change and the value is true/false but may change with further execution. For example, if we have a formula a->b and execute c we know for sure that we can never satisfy the formula (we say it is permanently false), whereas for -->b if we execute a c, the formula is only temporarily false (we still have proof obligations but may be able to satisfy them in the future). This allows us to terminate early once a formula is permanently true or false. This has the added advantage of allowing us to provide a rewritten formula after executing a sequence of steps, which often contains hints of shortcomings of the model.

The engine for testing BTL is used in 3 different tools. The grader discussed in Sect. 2 is used by teachers to finally grade assignments. Additionally, we provide two tools for testing BTL formulas (without grading). One is used to help a teacher create BTL specifications by providing immediate feedback on whether a CPN model (created by the teacher as a sample solution or given to the teacher) satisfies a BTL formula. Figure 5 shows a screenshot of testing the formula A-->C for the example in Fig. 4. The tool allows to manually create and step through a run of the CPN model, thereby observing how the BTL formula tested for is evaluated step by step. The panel Enabled Bindings shows the list of currently enabled transitions and bindings from

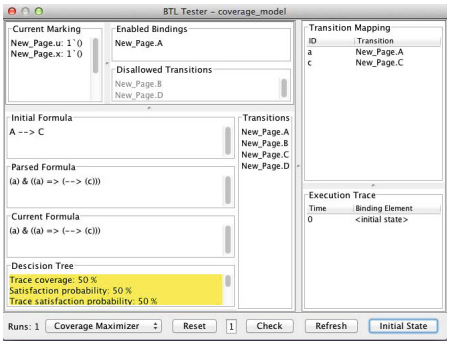


Fig. 5. Screenshot of BTL tester

which the user can pick one. Disallowed Transitions shows transitions not allowed by the guard function. The panel Current Marking shows the current marking in the run. The panel Current Formula contains the remaining formula that has to be evaluated, whenever a step in the CPN model makes a subformula true or false, the formula in that panel is rewritten according to the BTL semantics of Sect. 3. The panel Execution Trace shows the steps of the run executed so far, including timing information which is valuable for assessing whether time-related guards in the model match time-related conditions in the BTL formula. The tool also reports estimates of the coverage, probability the formula holds, and the probability of a random trace satisfying the formula in the Decision Tree panel.

A simplified version of this tool allows students to check that their models conform to the formulas. Here, the tool is pre-packaged with a set of BTL formulas that the model must satisfy. The students loads their models and the tool automatically tests validity of each formula on the model. In case one formula is not satisfied by the model, the student can manually single-step through the model and watch the formula progress in an interface similar to Fig. 5, aiding in finding and fixing obvious errors before handing in.

5.2 Case Study: Business Information Systems

In this section, we present first experiences we made with Grade/CPN in supporting the evaluation of a CPN assignment in the course Business Information Systems at Eindhoven University of Technology. In this assignment, students were given the base model in Fig. 1 and they had to model the delivery system according to a textual specification. Each of the 94 students had to work on five tasks; for each task, they had to submit one model. We received in total 258 models from 66 students. Table 1 summarizes some statistics. We continue by describing the assessment in more detail and then report on the experiences had.

Assessment. For each of the five tasks, the assessment consisted of two steps. In the first step, we applied Grade/CPN by calling it with a student model, the base model, and a configuration file (see Listing 2). Here, we were interested whether the interface and declaration of the base model have been preserved, whether there is a suspicion of fraud, and whether, depending on the task, six up to fourteen scenarios can be replayed on the model (only two are shown in the Listing). The scenarios were part of the specification of the assignment,

Table 1. Results of supporting the evaluation of 258 CPN models

Task	hand-ins	incorrect models	grader incorrect	full points by grader	full points
1	66	8	0	58	58
2	64	8	0	56	56
3	56	49	7	2	6
4	41	32	4	6	6
5	31	20	2	0	0

and we specified them using BTL. As BTL refers to the interface, it is crucial that students have not changed it. The runtime of the tool was about ten minutes for all students in the case of Task 1 and 2 and about ten minutes for each student model in the case of Tasks 3–5. The reason for the different runtime of Grade/CPN is that Tasks 1 and 2 are simple CPNs with few tests, whereas the remaining tasks performed more thorough tests on more advanced models, including performance analysis, thereby causing a higher analysis effort. Grade/CPN detected two fraud attempts, though they turned out to be caused by students handing in a subsequent assignment using the same base model. In a subsequent run of the course, we caught two students cheating, even though they had tried to conceal that by changing the layout. This attempt would be unlikely to get caught manually, but after singling out the models, we manually inspected them and saw they were clearly the same despite the obfuscation.

In the second step of the assessment, we manually checked each of the generated reports. On average, this took less than five minutes for each report in the case of Task 1 and 2 and about ten minutes in the case of Tasks 3–5. Based on the feedback provided by Grade/CPN, it was easy to check whether a model was actually correct or not, in particular for the untimed CPN models. Basically, the violation of a certain scenario simplified the detection of the cause for this violation drastically. In most cases, we did not even have to look at the counterexample provided by our tool. For Tasks 1 and 2, we had to simulate only five out of 130 student models manually to determine the cause of an error. A similar number of models had to be simulated manually for each of the Tasks 3–5. In those cases, the effort spent on finding the cause of an error was often higher because of the complexity of the models.

The tool automatically detected several subtle errors, such as wrong guards and minor changes to the environment, without having the need to manually open the respective model; it is highly unlikely we would have caught all of these completely manually. We even found subtle errors in our own solutions, yielding better results.

Experiences and Evaluation. Based on experience from previous years, the use of Grade/CPN reduced the amount of time for grading the assignment by a factor of at least two to three. This is factoring in that we used Grade/CPN for the first time and had to both define and understand the defined logic BTL, and also did not place complete confidence in the reported results which probably increased the manual labor as well. Table 1 confirms this observation: For each task, it shows the number of student models received (Col. 2), the number of incorrect models (Col. 3), how many times Grade/CPN gave incorrect results (Col. 4), the number of student models that were graded to be correct according to the tool (Col. 5), and the number student models that were graded to be correct after manually checking them (Col. 6). In fact, whenever the grader assigned full points to a model, then the model was correct. As a result, checking those models manually took almost no time. Given the high number of models for Tasks 1 and 2, we saved a lot of time here. Column 4 shows that only few models were graded incorrectly. In most cases, the cause was a misinterpretation of the

specification on the part of the students where we decided that the students should not be punished. Note that we do not show incorrect results of the tool caused by problems specifying a scenario in BTL.

The second column shows that the number of students participating at the assignment decreased from 66 for Task 1 to 31 for Task 5. Moreover, the average number of incorrect student solutions increased from 8/66=12% for Task 1 to 20/31 =65% for Task 5. The tasks became more difficult; whereas the first two tasks dealt with untimed CPN models and simple functionality, the remaining three tasks were much more involved. However, for the last two tasks we provided students with a student version of Grade/CPN. The idea was to provide them with a BTL specification that covers the basic functionality of their model. The final BTL specification used by us to grade their assignment contained additional scenarios. We experienced that providing students with Grade/CPN helped them to come up with better models. Whereas only $32/56 = 57\%$ of the students got at least half of the points for the third assignment ($56 - 49 = 7$ correct solutions), this number increased to $28/41 = 68\%$ (11 correct) for Task 4 and $20/31 = 64\%$ (11 correct) for Task 5 even though these were much more involved than the previous tasks. Moreover, we observed that the overall quality of the models increased drastically.

Grading models is a rather monotonous work. Therefore, it is easily possible that one oversees an error or forgets to check some scenario. Using Grade/CPN, this is now impossible and, therefore, we think that we can provide students with a fairer (in the sense of more equal) grading on the one hand and better feedback on the other hand.

Coverage Criteria and Confidence. We also compared the quality of the test result under the 3 different testing strategies (random exploration, increasing coverage of the guided tree, increasing transition coverage) discussed in Sect. 4. We observed that random yields the least confidence in the validity of the formula. Increasing tree coverage raises coverage by factor 4 (compared to random) and increasing transition coverage raises coverage by factor 200 (compared to random). Likewise, increasing the number of runs tested for also raises covarage.

6 Conclusion and Future Work

We have presented Grade/CPN, a tool to semi-automatically grade CPN models. Using Access/CPN, we can support any model created using CPN Tools. The plug-in architecture makes the tool easily extendible: to do so, one must just implement the interface in Listing 1 (ll. 1–5). The pluggable configuration with a very simple base format makes configuration simple. Configuration comprises selecting which plug-ins to use, which weight to assign them, and which parameters to instantiate them. Each plug-in only needs to consider its own options as the overall configuration format is handled by Grade/CPN. Reporting is handled by making all plug-ins return simple messages optionally annotated with more detailed reasoning (Listing 1 ll. 13–15). The information is automatically gathered by Grade/CPN and presented both as an overview in the user interface and as a detailed report. We have presented both simple plug-ins and a very powerful one implementing guided

checking of Britney Temporal Logic (BTL). BTL allows us to guide the simulation toward desired scenarios and to check that the environment contracts are adhered to. All plug-ins provide categorized information explaining the score and highlighting any changes made to the model, so teachers processing the reports only have to focus on things that cannot be automatically checked. We have designed and implemented an infrastructure for detecting fraud. We have reported on our experience with the Business Information Systems course where Grade/CPN was used to grade 258 assignments from 94 students. Using Grade/CPN instead of a completely manual approach reduced the manual labor by a factor of two to three. Grade/CPN is being employed again in the same course and results show that the quality of student models has significantly increased after giving them access to the student tester.

The idea of (semi-)automatically grading assignments is not new and closely related to testing. A known testing framework is JUnit [14], which also runs a set of tests and reports the result. The advantage of our tool over JUnit is that JUnit requires programming to get started, whereas we use simple configuration files. From the testing world we also find the tool Jenkins (previously Hudson) [15], which runs tests on a central server and provides near-instantaneous feedback. The main disadvantage of Jenkins in our view is also complexity; while it does not (necessarily) require programming, setup does require complex XML configuration, and extension either requires huge effort or makes it difficult to get consolidated reports. There are many tools for automatically grading programming assignments [16], for example, the tool peach^3 [17], which more focuses on managing hand-ins, but can also run automatic tests. In contrast, we focus on the tests and CPN models directly and assume that models already exist. Our testing approach is similar to runtime LTL [10,13], but our logic also supports guiding. This is similar to hot/cold events in Live-Sequence Charts [18], but our sections are more urgent in that a guide is not only preferred, it is an immediate failure if it is not possible to follow it, making BTL computationally easier to check.

It is very interesting to increase the efficiency of the coverage heuristics for BTL, including expression abstraction, e.g., using a Counter-Example Guided Abstraction Refinement (CEGAR) [19] or similar approach. It is also interesting to employ more static analysis to get even better coverage. Experience says, though, that students often fail to account for particular cases, making it very easy to detect errors in those cases. It would also be interesting to investigate simpler languages. For example, it may be interesting for a teacher simply to see if a given transition is enabled. This is easily expressible in BTL but difficult to check, and employing techniques from directed model-checking [20] may prove beneficial to try more intelligent guiding towards errors. We would also like to extend Grade/CPN with ability to provide simple simulation-based checks of standard safety and liveness properties. We also want to add support for loading models in the PNML standard [21] format to be able to also check models created using other tools.

Acknowledgements. The authors thank Boudewijn van Dongen for fruitful discussions about the requirements for an automatic grader.

References

1. Jensen, K., Kristensen, L.M.: Coloured Petri Nets – Modelling and Validation of Concurrent Systems. Springer (2009)
2. van der Aalst, W.M.P., Stahl, C.: Modeling Business Processes – A Petri Net-Oriented Approach. MIT Press (2011)
3. Online: CPN Tools webpage, `http://cpntools.org`
4. Westergaard, M., Kristensen, L.M.: The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 313–322. Springer, Heidelberg (2009)
5. Westergaard, M., Fahland, D., Stahl, C.: Grade/CPN: Semi-automatic Support for Teaching Petri Nets by Checking Many Petri Nets Against One Specification. In: Proc. of PNSE. CEUR Workshop Proceedings, vol. 851, pp. 32–46. CEUR-WS.org (2012)
6. Westergaard, M., Evangelista, S., Kristensen, L.M.: ASAP: An Extensible Platform for State Space Analysis. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 303–312. Springer, Heidelberg (2009)
7. Pnueli, A.: The Temporal Logic of Programs. In: Proc. of SFCS 1977, pp. 46–57. IEEE Comp. Soc. (1977)
8. Kripke, S.A.: A semantical analysis of modal logic: I. Normal modal propositional calculi. Zeitschrift für Mathematische Logic und Grundlagen der Mathematik 9, 67–96 (1963)
9. Plotkin, G.: A Structural Approach to Operational Semantics. DAIMI-FN 19, Department of Computer Science, University of Aarhus (1981)
10. Giannakopoulou, D., Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs. In: Proc. ASE, pp. 412–416. IEEE Computer Society (2001)
11. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers (2006)
12. Weißleder, S.: Simulated satisfaction of coverage criteria on uml state machines. In: ICST 2012, pp. 117–126 (2010)
13. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL Semantics for Runtime Verification. Logic and Computation 20(3), 651–674 (2010)
14. Online: JUnit webpage, `http://junit.org`
15. Online: Jenkins Continuous Integration webpage, `http://jenkins-ci.org`
16. Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O.: Review of Recent Systems for Automatic Assessment of Programming Assignments. In: Proc. International Conference on Computing Education Research, pp. 86–93. ACM (2010)
17. Verhoeff, T.: Programming Task Packages: Peach Exchange Format. Olympiads in Informnatics 2, 192–207 (2008)
18. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Form. Methods Syst. Des. 19(1), 45–80 (2001)
19. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement for Symbolic Model Checking. J. ACM 50, 752–794 (2003)
20. Edelkamp, S., Lluch Lafuente, A., Leue, S.: Directed Explicit Model Checking with HFS-SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 57–79. Springer, Heidelberg (2001)
21. ISO/IEC: Software and system engineering – High-level Petri nets – Part 2: Transfer format. ISO/IEC 15909-2:2011