Grade/CPN: Semi-automatic Support for Teaching Petri Nets by Checking Many Petri Nets Against One Specification

Michael Westergaard, Dirk Fahland, and Christian Stahl

Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
{m.westergaard,d.fahland,c.stahl}@tue.nl

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 automatize, 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) [8] is a formalism useful for modeling a broad range of real-life systems, including complex network protocols [8] and business information systems [1]. 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. As we aim at supporting grading for all kinds of models, we here focus on the testing perspective, as a model may not be suitable for 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 analysis of 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.

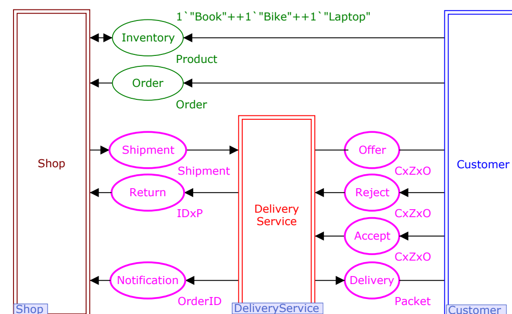


Fig. 1: Base model of a delivery service.

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. 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 practise, 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 facilitating this. 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 to 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.

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 [14] as it provides an easy way to load CPN models and programatically interact with the simulator.

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, and
   (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, and
7. Provides a report that pin-points problems, aids the teacher in grading, and allows students to understand problems.

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 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.

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 5 is handled partly in this section and in the next section, where we also deal with requirement 4(b) (checking dynamic properties).

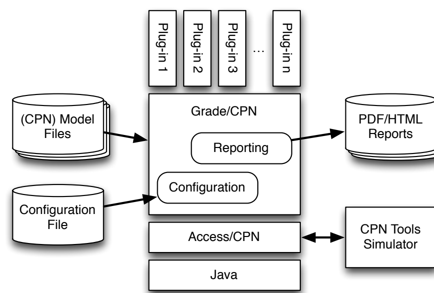


Fig. 2: Overall architecture and environment of Grade/CPN.

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 [14]. 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 interfaced 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 `PlugIns`).

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.

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 }
```

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.

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-

Listing 2: Example configuration file.

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

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

in (l. 5) makes sure that no declaration in the provided model is removed or 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 compare the student model with the base model manually.

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 false, which means that students are not allowed to add new pages. The model used here is flat, and thus introducing hierarchy is considered an error. 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).

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 simulator; that is, we apply a testing-based approach rather than exhaustive state-space exploration, yielding a sound but not necessarily complete checking mechanism.

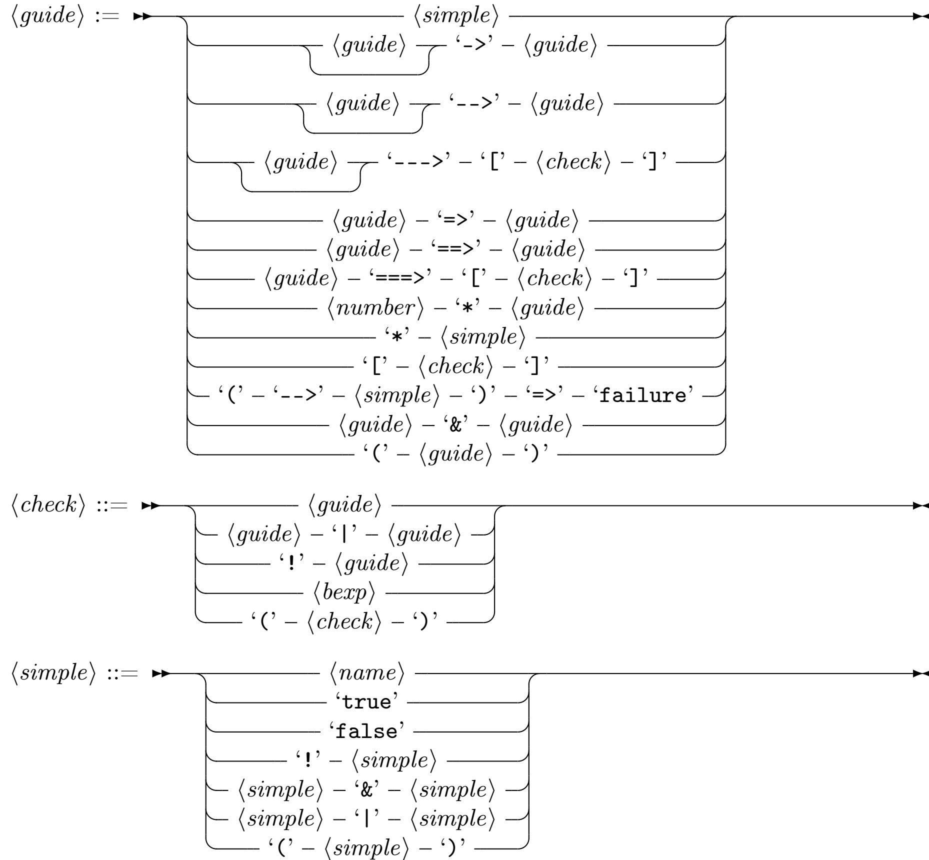
Guiding the simulator requires to specify which transition the simulator should execute. Testing whether some property holds in a state of the model requires a specification of this property. To this end, we introduce the *Britney Temporal Logic* (BTL). This logic is similar to linear-time logic (LTL) [12] but in addition to checking properties also allows guiding the simulator and to specify constraints that should hold in a state. We also 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. In the following, we define the syntax of BTL formulae and then their semantics based on Kripke structures [10] and structural operational semantics (SOS) [11].

3.1 Syntax

A BTL formula is a $\langle guide \rangle$. A guide describes how simulation should be performed; that is, it guides the simulator to a desired state. The atomic propositions of a guide are described using $\langle simple \rangle$, which is an expression without temporal operators but otherwise allowing full propositional logic on transitions and place invariants. The temporal operators are various 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 at some point 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 kind of arrow, we also add a double arrow version indicating that *if* a holds, then b just holds at the appropriate time. 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. We allow conjunctions of guides using &, but no disjunctions because for an expression like $(a\texttt{->}b)|(c\texttt{->}d)$ it is not obvious whether to guide the simulation with an a or a c if both are enabled (as we do not know whether b or d are enabled in the next step).

A guide can also include ⟨*check*⟩s, which are not used to guide the simulator but only to test assertions. They are therefore allowed to contain disjunctions and negations and general boolean expressions. Finally, a guide can include the special keyword '`failure`', which is a synonym for '`false`' but with a very restricted syntax. This means that we try to stay clear of transitions that would violate the formula. The BTL formula in Listing 2 (ll. 8-12) guides a model to execute a transition Order exactly 10 times (l. 9) and a transition Receive exactly 10 times (l. 10) with any intermediate transitions allowed except for Reject (l. 11).



In addition to the syntax for guiding, we also allow boolean expressions. These are mostly for testing state properties, such as counting the tokens on a

place or testing values of the global clock. Boolean expression are defined in the standard way and shall not be repeated here due to space limitations. 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 3 or more tokens in total.

3.2 Semantics

We interpret formulae specified in BTL over the state space of a CPN. The state space of a CPN can be seen as 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. As BTL is an LTL-like logic, we also introduce the notion of a trace. A trace is a transition sequence $q_0, \ldots, q_k$ such that q_0 is the initial state and for all $0 \leq i < k$, there exists an $a \in \Sigma$ with $(q_i, a, q_{i+1}) \in \delta$. The semantics is similar to a standard finite trace semantics for LTL like the one defined in [5].

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* and that '`failure`' is semantically the same as '`false`'. 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:

$$\begin{aligned}
\texttt{->}G &\equiv \mathit{true}\texttt{->}G \\
\texttt{-->}G &\equiv \mathit{true}\texttt{-->}G \\
\texttt{--->}[B] &\equiv \mathit{true}\texttt{--->}[B] \\
\mathit{failure} &\equiv \mathit{false} \\
G_1\texttt{->}G_2 &\equiv G_1 \& G_1\texttt{=>}G_2 \\
G_1\texttt{-->}G_2 &\equiv G_1 \& G_1\texttt{==>}G_2 \\
G\texttt{--->}[B] &\equiv G \& G\texttt{===>}[B] \\
G_1\texttt{==>}G_2 &\equiv G_1\texttt{->}(*\mathit{true}\texttt{->}G_2) \\
n * (G) &\equiv \begin{cases} G\texttt{->}(n-1) * (G) & \text{if } n \geq 1 \\ \mathit{true} & \text{otherwise.} \end{cases}
\end{aligned}$$

The semantics of $\langle \mathit{simple} \rangle$ can now be defined over the traces of a Kripke structure K. Attribute *name* in the grammar thereby refers to a transition label.

- Every trace $(q_0, \ldots, q_k)$ satisfies $(q_0, \ldots, q_k) \models \mathit{true}$ and $(q_0, \ldots, q_k) \not\models \mathit{false}$.
- $(q_0, q_1, \ldots, q_k) \models \mathit{name}$ iff $q_0 \xrightarrow{\mathit{name}} q_1$.
- $(q_0, q_1, \ldots, q_k) \models !S$ iff $(q_0, q_1, \ldots, q_k) \not\models S$.
- $(q_0, q_1, \ldots, q_k) \models S_1 \& S_2$ iff $(q_0, q_1, \ldots, q_k) \models S_1 \wedge (q_0, q_1, \ldots, q_k) \models S_2$.

– $(q_0, q_1, \ldots, q_k) \models S_1|S_2$ iff $(q_0, q_1, \ldots, q_k) \models S_1 \vee (q_0, q_1, \ldots, q_k) \models S_2$.

Boolean expressions can be evaluated over the set AP of atomic propositions and are, therefore, evaluated at a state. The remaining operators are LTL-like and are therefore defined over traces. We write $q_0 \xrightarrow{G} q_k$ to denote that $(q_0, \ldots, q_k) \models G$. The first is unbounded repetition. We notice that this is satisfied regardless of what we do (as zero repetitions can be performed).

$$\frac{q \xrightarrow{G} q'', q'' \xrightarrow{(\text{`*'}G)} q'}{q \xrightarrow{(\text{`*'}G)} q'}, \quad \frac{}{q \xrightarrow{(\text{`*'}G)} q'} \tag{1}$$

Operator `=>` is similar to the next operator in LTL (though here it is conditional): If G_1 holds on a trace $q_0, \ldots, q_j$ then G_2 must hold starting from q_j.

$$\frac{q_0 \xrightarrow{G_1} q_j \implies q_j \xrightarrow{G_2} q_k, 0 < j < k}{q_0 \xrightarrow{G_1\texttt{=>}G_2} q_k} \tag{2}$$

The following three rules define operators used for checking a property (expressed by putting the expression into squared brackets). We use them to check whether a boolean expression holds, a boolean expression holds in a final state after guiding the simulator using expression G, and whether a guide holds.

$$\frac{q_0 \models B}{q_0 \xrightarrow{[B]} q_k}, \quad \frac{q_k \models B}{q_0 \xrightarrow{G\texttt{===>}[B]} q_k}, \quad \frac{\forall j \leq k : \neg(q_0 \xrightarrow{G} q_j)}{q_0 \xrightarrow{G\texttt{===>}[B]} q_k}, \quad \frac{q_0 \xrightarrow{G} q_k}{q_0 \xrightarrow{[G]} q_k} \tag{3}$$

Finally we define a conjunction as usual:

$$\frac{q_0 \xrightarrow{G_1} q_k \wedge q_0 \xrightarrow{G_2} q_k}{q_0 \xrightarrow{G_1\&G_2} q_k} \tag{4}$$

The final consideration is how we guide. This is done by defining a set of allowed transitions for each guide. For simulation, only 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). We define the set *guide* over a set T of possible transitions inductively as:

– $guide(q, S) = \{name \in T \mid q \xrightarrow{name} q' \implies (q, q') \models S\}$,
– $guide(q, n\text{`*'}G) \begin{cases} guide(G) & \text{if } n \geq 1, \\ T & \text{otherwise,} \end{cases}$
– $guide(*G) = T$,
– $guide(q, G_1\texttt{=>}G_2) = T$,
– $guide(q, [B]) = T$,
– $guide(q, G\texttt{===>}[B]) = T$,
– $guide(q, [G]) = T$,

- $\mathit{guide}(q, G_1 \wedge G_2) = \mathit{guide}(q, G_1) \cap \mathit{guide}(q, G_2)$,
- $\mathit{guide}(q, (\texttt{-->}S) \texttt{=>} \mathit{failure}) = T \setminus \mathit{guide}(q, S)$.

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

3.3 Implementation

Our implementation of BTL uses simple formula rewriting. 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-value logic similar to [2]. 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. Figure 4 shows such an error report, containing an error trace (which we have shortened here), the violated formula (ll. 10–14 from Listing 2), the formula at the error, and the marking at the error. If a model has no error, we instead show the final state which can be manually inspected for irregularities, e.g. improper termination.

```
• Executed Trace:
  System.Order{ c = "Jim", i = 0, p = "Book" }
  System.Ship{ c = "Jim", id = 0, p = "Book" }
  System.Order{ c = "Jim", i = 1, p = "Laptop" }
  System.Put_in_Storage{ c = "Ann", id = 8, p = "Book" }
  System.Put_in_Storage{ c = "Ann", id = 9, p = "Bike" }
• Initial Formula:
  (10 * (--> Order) -> (--> Order) => failure) &
  ((--> Reject) => failure) &
  (10 * (--> Receive) -> (--> Receive) => failure) &
  [(--> Handle_Return) => false]
• Formula at error:
  ((((--> (order)) => failure) & ((--> (reject)) => failure)) & ((10 * (--> (receive))) & ((10 * (-->
  (receive))) => ((--> (receive)) => failure)))) & ([(--> (handlereturn)) => (!(true))])
• Marking at error:
  System.Accept: 1`(1,"Jim")
  System.Count: 1`10
```

Fig. 4: Error report.

In addition to the grader used by teachers to finally grade assignments, we also have two tools for testing BTL formulas. One is used during construction to help a teacher get immediate feedback on a formula and a solution by manually or experimentally testing formulas on a proposed solution. A simplified version of this tool allows students to check that their models conform to the formulas. This tool also allows students to manually single-step through their model and watch the formula progress, aiding in finding and avoiding obvious errors before handing in.

4 Practical Experience

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 the 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 submit two models. We received in total 130 models from 66 students.

In a first step of the assessment, 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 six scenarios can be replayed on the model (only two are shown in the Listing). The scenarios were part of the specification of the assignment, 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; that is, after this time, a report had been generated for each student. The tool detected two fraud attempts, though they turned out to be caused by students handing in a subsequent assignment using the same base model as well.

In a second step, we manually checked each of the generated reports. On average, this took less than five minutes for each report. Based on the feedback provided by Grade/CPN, it was easy to check whether a model was actually correct or not. 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. Overall, we had to simulate only five models manually to determine the cause of an error. The tool automatically detected several subtle errors such as wrong guards and minor changes to the environment without having the need to 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.

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. As correcting models is a rather monotonous work, 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.

5 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 130 assignments from 66 students. Using Grade/CPN instead of a completely manual approach reduced the manual labor by a factor of three. Each student has to hand in a total of five models during the course, so anticipated time savings are immense.

The idea of (semi-)automatically grading assignments is not new and closely related to testing. A known testing framework is JUnit [9], which also runs a set of tests and reports the result. The advantage of our tool over JUnit is that JUnit requires programming to even get started, whereas we use simple configuration files. Also from the testing world is Jenkins (previously known as Hudson) [7], 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 [6], for example, the tool peach[3] [13], which more focus 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 [2, 5], but our logic also supports guiding. This is similar to hot/cold events in Live-Sequence Charts [4], 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. This makes BTL computationally easier to check.

Future work includes loosening what is allowed in guides by having the tool try resolving, e.g., disjunctions itself (by keeping track of which branches have been explored and only reporting errors if no branch is successful). We also consider a designer for automatically building BTL formulae and full configuration files by manually guiding the simulation in a manner similar to our current

tool for testing BTL formulas. We aim to integrate Grade/CPN with Jenkins or peach[3] so we can combine our simplicity of configuration with the more advanced features of those systems. While BTL is designed for grading assignments using testing, it has also proved useful for finding errors in our model. It would be interesting to investigate this further, including making testing complete by exploiting the guiding perspective of our logic.

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

References

1. W.M.P. van der Aalst and C. Stahl. *Modeling Business Processes – A Petri Net-Oriented Approach.* MIT Press, 2011.
2. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. *Logic and Computation*, 20(3):651–674, 2010.
3. CPN Tools webpage. Online: `cpntools.org`.
4. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Form. Methods Syst. Des.*, 19(1):45–80, 2001.
5. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proc. ASE*, pages 412–416. IEEE Computer Society, 2001.
6. P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proc. International Conference on Computing Education Research*, pages 86–93. ACM, 2010.
7. Jenkins Continuous Integration webpage. Online: `jenkins-ci.org`.
8. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems.* Springer, 2009.
9. JUnit webpage. Online: `junit.org`.
10. S.A. Kripke. A semantical analysis of modal logic: I. Normal modal propositional calculi. *Zeitschrift für Mathematische Logic und Grundlagen der Mathematik*, 9:67–96, 1963.
11. G.D. Plotkin. A Structural Approach to Operational Semantics. DAIMI-FN 19, Department of Computer Science, University of Aarhus, 1981.
12. A. Pnueli. The Temporal Logic of Programs. In *Proc. of SFCS '77*, pages 46–57. IEEE Comp. Soc., 1977.
13. T. Verhoeff. Programming Task Packages: Peach Exchange Format. *Olympiads in Informnatics*, 2:192–207, 2008.
14. M. Westergaard and L.M. Kristensen. The Access/CPN Framework: A Tool for Interacting With the CPN Tools Simulator. In *Proc. of ATPN*, volume 5606 of *LNCS*, pages 313–322. Springer, 2009.