Embedding Java Types in CPN Tools

K.B. Lassen and M. Westergaard

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: {k.b.lassen,mw}@daimi.au.dk

Abstract. CPN Tools is a well known editor for Colored Petri nets (CPNs) that is capable of doing state space and performance analysis. The BRITNeY Suite has added yet another feature to CPN Tools for integrating CPN models with Java programs, by providing stubs accessible from the models, to allow the modeller to call methods on Java objects. This paper is about how the stub code is generated, i.e., representing Java classes to Standard ML to be able to call Java code in the CPN models, and how the BRITNeY Suite framework handles the invocations of the stub code. The contribution of this paper is give an in depth understanding of how CPN Tools and BRITNeY Suite work together.

1 Introduction

When making models in CPN Tools [2, 8] it may be convenient to interact with other tools for instance to visualise various aspects of a coloured Petri net (CPN or CP-net) [4] model. In [6] a protocol for communication between mobile ad-hoc nodes in a network was constructed and visualised for presentation to military leaders with no knowledge of formal methods. This means that besides having the CPN model to analyse for standard properties, the model could be validated against an informal protocol description by experts that had domain knowledge but no knowledge of CP-nets. Here the domain experts, where capable of saying that what was modelled actually was the desired system or not. This was made possible by making the CPN Tools interact with the BRITNeY Suite [10, 11]. In [5], a model the work-flow of a blanc loan application was visualised, so bank assistants was able to validate the behaviour of the model. Again, with the help of the BRITNeY Suite.

This paper describes how it is possible to call Java code, during simulation or state space analysis, from a CP-net model, within CPN Tools. It is not a trivial problem, because it involves mapping concepts from the object oriented world of Java, into the functional world, i.e. CPN ML (the inscription language of CPN Tools). In this paper we assume that the reader has a good understanding of Standard ML (SML) [7], object oriented programming, preferably Java, as well as CPN Tools.

The paper is structured as follows: Section 2 presents an architectural overview of the communication between CPN Tools and BRITNeY Suite with focus on

how values are being passed through back and forth. In Sect. 3 we give an overview of how to access Java code from CPN Tools. Section 4 describes a crude approach to representing different Java types in SML. Section 5 presents how we can handle a larger subset of Java and CPN ML, than is made possible with the crude approach from Sect. 4. Finally, in Sect. 6 we conclude.

2 Architectural Overview

Figure 1 shows the overall architecture of BRITNeY Suite when it is being run in conjunction with CPN Tools. On the left we have the CPN Tools GUI, which communicates through the BRITNeY Suite, in the middle, with the CPN Simulator, on the right. We have zoomed in on a couple of the internal components of the BRITNeY Suite, and we can see that it contains a Simulator proxy, which mediated communication between the CPN Tools GUI and the simulator. BRITNeY Suite has some Animation plug-ins, which contains the functionality we want to export to the CPN Simulator. The CPN Simulator communicates with these animation plug-ins using simple remote procedure call (RPC) [1, Chap. 5.3]. In order to this to work, we need to have some stubs in the simulator and these are generated by the Stub generator component. In this paper we will focus on the Stub generator.

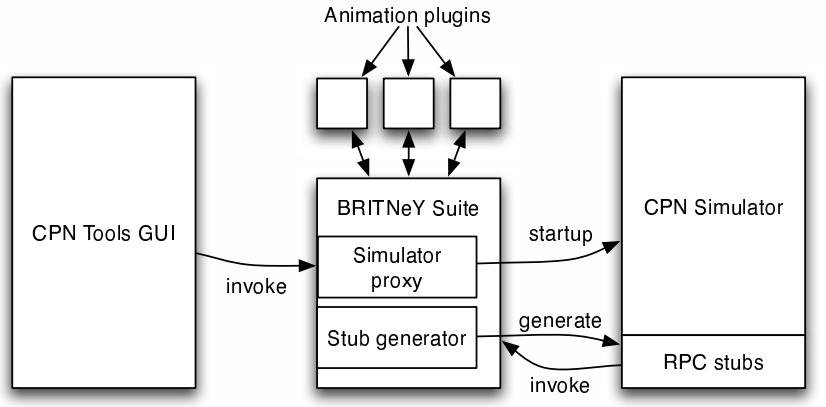

Fig. 1. Overview of the architecture of BRITNeY Suite and CPN Tools.

When the stub-generator has generated the stub code, we need to evaluate it in the simulator so that it can be used from the CPN Tools GUI. As the code needs to be available before the CP-nets is checked, we need to do this before the CPN Tools GUI gains control of the simulator. In order to accomplish this, the simulator proxy takes the place of the normal CPN Simulator. In Fig. 2 we see how this is performed. When the CPN Tools GUI requests a simulator, the Simulator proxy of BRITNeY Suite intercepts this and starts an actual

CPN Simulator. Then the Stub generator takes control, generates stub code, as described in the following sections. This code can then be evaluated in the real CPN Simulator, controlled by the BRITNeY Suite, and the control can be returned to the CPN Tools GUI (actually the control is not returned completely, but all calls from the CPN Tools GUI goes through the BRITNeY Suite, which becomes useful in Sect. 5).

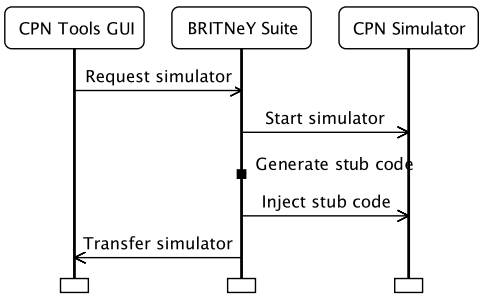

Fig. 2. Abstract view of how BRITNeY Suite injects code into the simulator before the CPN Tools GUI gains control over it.

3 The Stub Generator

The generated stub-code is actually quite simple. We make it possible to access objects in two ways. Either directly (think of a static instance in Java), or indirectly by using strings as object references.

Suppose we have implemented a plug-in, which provides a phone-book service. The Java interface for our service may look like the one provided in Listing 1. The actual implementation of this interface is outside the scope of this paper. The interface contains methods for looking up the name of the owner of a phone number (l. 2), for looking up all numbers under a given address and given names (l. 3). We assume that each number has exactly one owner. The interface also has a method to add a new person to the data base (l. 4). This method uses the PersonInterface described in lines 9–21. Finally the interface has a method to return all data about all persons on a given address (l. 5) as well as a method to add an arbitrary note to a phone number (l. 6). The PersonInterface contains methods to set and get the name of the person (ll. 10–11), the address of the person (ll. 13–14), all numbers owned by the person (ll. 16–17), and to categorize the person (ll. 19–20) using the Java enumeration type PersonType defined in line 23.

For an animation plug-in PhoneBook implementing the PhoneBookInteface interface, the stub generator generates the code in Listings 2 and 3. The code

Listing 1 A Java-interface for a phone-book service.

```
1  public interface PhoneBookInterface {
2      String getOwnerName(int number) throws NumberNotFound;
3      int[] lookupNumbers(String names[], String address);
4      void addPerson(PersonInterface p);
5      PersonInterface[] getPersons(String address);
6      void addNote(int number, Object note)
7  }
8
9  public interface PersonInterface {
10     String getName();
11     void setName(String name);
12
13     String getAddress();
14     void setAddress(String address);
15
16     int[] getNumbers();
17     void setNumbers(int[] numbers);
18
19     public PersonType getType();
20     public void setType(PersonType type);
21 }
22
23 public enum PersonType { Male, Female, Child };
```

contains two more or less equal parts, namely one for indirect, reference-based access, Listing 2, and one for direct, static, access, Listing 3. Let us first look at the reference based version in Listing 2.

Lines 1–13 contain a signature for the plug-in. A signature in SML corresponds to an interface in Java. We see that every implementation of PhoneBook in SML have an exception of type string (l. 3). This exception is raised if something goes wrong (communication failure, failure in the Java code, etc., such as lookup of a non-existent number in getOwnerName). The exception contains an error message describing the error. All generated stubs will contain such an exception.

Lines 5 states that there is a function create_instance, which creates a new instance of the PhoneBook plug-in and returns an object reference to the newly created object.

Lines 7–12 are signatures for methods in the Java class that we create the stub for. All functions takes a string (the object reference returned by create_instance) and returns a function with parameters corresponding to the Java method. How these types are generated will be explained in greater detail in Sect. 4.

Lines 15–23 consists of a functor implementing the PHONEBOOK'NO_INSTANCE signature. The functor is parametrised by a host and a port that is used to specify where the BRITNeY Suite is running (as it may not be a local process). Sections 4 and 5 detail how the pseudo code <implementation of method X> is

actually generated.

In lines 25–27 we create a shortcut to the generic functor created above, where we hardwire the host and port to the local machine. This makes is possible to use PhoneBookRemoteNoInstance by simply writing structure phonebook = PhoneBookNoInstance() in CPN Tools.

Listing 2 Generic representation of a Java class named PhoneBook, reference based version.

```
1  signature PHONEBOOK'NO_INSTANCE =
2  sig
3      exception Exception of string
4
5      val create_instance : string -> string
6
7      val getOwnerName: string -> int -> string
8      val lookupNumbers: string -> string list * string -> int list
9      val addPerson: string ->
10         { name: string, address: string, numbers: int list } -> unit
11     val getPersons: string -> string ->
12         { name: string, address: string, numbers: int list } list
13 end
14
15 functor PhoneBookRemoteNoInstance(val host: string val port: int)
16     :> PHONEBOOK'NO_INSTANCE =
17 struct
18     <implementation of create_instance>
19     <implementation of getOwnerName>
20     <implementation of lookupNumbers>
21     <implementation of addPerson>
22     <implementation of getPersons>
23 end
24
25 functor PhoneBookNoInstance():> PHONEBOOK'NO_INSTANCE =
26     PhoneBookRemoteNoInstance
27         (val host = "localhost" val port = 11001)
```

Listing 3 is more or less the same as Listing 3. The only difference is we no longer are able to create instances dynamically, so the create_instance function has been removed from the signature, a new reset method has taken its place (l. 5), and the generated signatures (ll. 7–12) no longer contain the first string parameter.

The implementation of this signature (ll. 15–29) uses the PhoenBookRemoteNoInstance functor (ll. 18–19) by calling create_instance on creation (l. 21) and automatically passing this value in all functions (ll. 25–28). The reset function simply renews the stored object reference (l. 23), which is useful for resetting the animation plug-in prior to e.g. a new simulation. If the animation plug-in

provides its own reset method, the default one, as shown in this example, is not created, but a stub similar to the other functions is generated in its place. This functor and the PhoneBook functor, are present as a convenience to the user, who does not need to dynamically create new instances of animation plug-ins.

Listing 3 Generic representation of a Java class named PhoneBook, static instance version.

```
1  signature PHONEBOOK =
2  sig
3      exception Exception of string
4
5      val reset : unit -> unit
6
7      val getOwnerName: int -> string
8      val lookupNumbers: string * string -> int list
9      val addPerson:
10         { name: string, address: string, numbers: int list } -> unit
11     val getPersons: int ->
12         { name: string, address: string, numbers: int list } list
13 end
14
15 functor PhoneBookRemote
16     (val host: string val port: int val name: string):> PHONEBOOK =
17 struct
18     structure Instance =
19         PhoneBookRemoteNoInstance(val host = host val port = port)
20
21     val reference = ref (Instance.create_instance name)
22
23     fun reset () = reference := Instance.create_instance name
24
25     val getOwnerName = Instance.getOwnerName (!reference)
26     val lookupNumbers = Instance.lookupNumbers (!reference)
27     val addPerson = Instance.addPerson (!reference)
28     val getPersons = Instance.getPersons (!reference)
29 end
30
31 functor PhoneBook(val name: string):> PHONEBOOK =
32     PhoneBookRemote
33         (val host = "localhost" val port = 11001 val name = name)
```

We assume that all Java classes exported as animation plug-ins have at least a constructor that takes a string. This value is passed name value in lines 16 and 31 in Listing 3 or as the argument to the create_instance function in line 5 of Listing 2.

4 Simple Types, Arrays, and Java Beans

In this section we describe how methods of Java classes are represented in SML. First, we describe how we represent every Java method, without looking at the type of input or output it produces, this is dealt with later on. In Sect. 4.2 we show how we handle simple types, Sect. 4.3 describe how we handle arrays, and Sect. 4.4 shows how classes adhering to the Java Beans [3] programming paradigm are handled.

4.1 Method representation

In order to make the method implementation as simple as possible, we have loaded a runtime system into the CPN simulator on startup. The runtime system consists of a structure, Runtime, which exposes a single method, invoke, which takes a name of a method and a list of untyped parameters. It returns an untyped parameter.

Each method in a Java class is represented as in Listing 4; we assume that the method that is represented in the listing is B methodName(A1 x1, A2 x2, ..., An xn) that belongs to Klass. The code contains the declaration of a method methodName which takes an object, the object reference, and a tuple of the arguments, (Klass'x1, Klass'x2, ..., Klass'xn) (l. 1). The method calls the invoke method of the Runtime (ll. 2–4) with the first parameter a fully qualified name of the object (l. 3) and the second parameter, a list of all parameters. We return to the concrete implementation of ¡argument n¿ later on. If the return argument is received properly, the returned value is type checked and returned as the methodName function result (l. 5). Otherwise, the function will raise an exception (l. 6). If this exception is ever thrown it indicates an exceptional case where the stub and method signature do not match.

Listing 4 Pseudo code representation of Java method in SML.

```
1 fun methodName object (Klass'x1, Klass'x2, ..., Klass'xn) =
2     (case Runtime.invoke(
3         object ^ ".methodName",
4         [<argument 1>, ..., <argument n>]) of
5      <return value unparsed> => <return value parsed>
6           | _ => raise Runtime.Exception "Bad return type")
```

The actual implementation of ¡argument n¿ and ¡return value unparsed¿ use the syntax structure in Listing 5 which is part of the runtime system. The param union data-type (ll. 3–7) is used to represent untyped values.

4.2 Simple Types

Simple types are very easy to represent in SML as they have a near one-to-one correspondence to Java types. Simple Java types that we can handle are

Listing 5 Structure with data-types used by the runtime system.

```
1  structure syntax
2  = struct
3      datatype param = STRING of string | INT of int | DOUBLE of real
4                     | BOOLEAN of bool | DATE of string
5                     | BASE64 of string | UNIT of unit
6                     | ARRAY of param list
7                     | STRUCT of (string * param) list
8
9      datatype method = METHOD of string * param list
10                     | FAULT of param
11 end
```

String, int, double, boolean, Date, Base64 and void. These are represented as syntax.STRING, syntax.INT, syntax.DOUBLE, syntax.BOOLEAN, syntax.DATE, syntax.BASE64 and syntax.UNIT in SML.

Listing 6 shows how the String getOwnerName(int number) method on line 2 in Listing 1 is represented. Here we have one parameter in addition to the object reference (l. 1), we generate a method name depending on the object reference and the getOwnerName method name (l. 3), we create an untyped parameter containing an integer (l. 4) and expect an untyped result containing a string (l. 5).

Listing 6 Simple types example.

```
1  fun getOwnerName object (PhoneBook'x1) =
2      (case Runtime.invoke(
3          object ^ ".getOwnerName",
4          [syntax.INT PhoneBook'x1]) of
5          (syntax.STRING PhoneBook'x0) => x0
6           | _ => raise Runtime.Exception "Bad return type")
```

4.3 Arrays

We have chosen to represent arrays as lists in SML. Another choice could have been to use the Array structure that SML/NJ provide, but because more people use lists and they are more natural in a functional language, we decided to use them instead. In the syntax.param data-type we find arrays as syntax.ARRAY, which contains a list of untyped values.

The example in Listing 7 we see how the int[] lookupNumbers(String names[], String address) on line 3 in Listing 1 is represented. Lines 1–3 should be familiar. In lines 4–7 we encode the first parameter, the list of integers. We want a syntax.ARRAY (l. 4). We convert a list of integers to a list of untyped values using the List.map function (ll. 5–7), which takes a function mapping integers to

the corresponding untyped value (l. 6) and a list of integers (l. 7). We pass the second value as a untyped string (l. 8) and expect an array back (l. 9), which we unwraps in lines 10–12.

Listing 7 Array representation example.

```
1  fun lookupNumbers object (PhoneBook'x1, PhoneBook'x2) =
2      (case Runtime.invoke(
3          object ^ ".lookupNumbers",
4          [syntax.ARRAY (
5              List.map
6              (fn PhoneBook'x3 => syntax.INT PhoneBook'x3)
7              PhoneBook'x1,
8           syntax.STRING PhoneBook'x2)]) of
9      (syntax.ARRAY PhoneBook'x0) =>
10         List.map
11             (fn (syntax.INT PhoneBook'x4) => PhoneBook'x4)
12             PhoneBook'x0
13     | _ => raise Runtime.Exception "Bad return type")
```

4.4 Java Beans

In Section 4.2 we showed how to handle simple types in Java. These where types that we immediately could map to SML. We can also represent the state of more general classes, namely classes living up to the Java Beans coding standard. This basically states that for each field in the class, we must have a getter and a setter method. For example, a class Person, implementing the PersonInterface in lines 9–21 in Listing 1 will live up to this coding style, and contain three fields: name, address, and numbers. We cannot at this point understand the third field, type, as it uses the type PersonType, which is neither a simple type, a list, nor a Java Bean, so let us disregard the details of how this field is handled until Sect. 5.3. We will let Java Beans correspond to records in SML, where each field of the record correspond to a field in the Java class.

The Person class can be represented in CPN ML as the colour definition in Listing 8; we assume that the colours STRING, INT, and INT_LIST have already been defined as string, int, and INT list respectively.

Listing 8 CPN-ML representation of the Person Java Bean.

```
1  colset Person = record
2      name : STRING *
3      address : STRING *
4      numbers : INT_LIST *
5      type : ...
```

In Listing 9 we see how the method void addPerson(PersonInterface p) is represented. Now the method header becomes a bit more elaborate, as we request a record (ll. 2–5). We translate the record into the untyped value using syntax.STRUCT (ll. 8–12), where each entry in the record is represented as a pair of the name as a string and the untyped value. We have omitted the actual representation of the numbers parameter in line 11, as it is just a wrapping like in Listing 7 lines 4–7.

Listing 9 Java Bean representation example.

```
1  fun addPerson object
2      ({address = PhoneBook'x1,
3        name = PhoneBook'x2,
4        numbers = PhoneBook'x3,
5        type = PhoneBook'x4}) =
6      (case Runtime.invoke(
7          object ^ ".addPerson",
8          [syntax.STRUCT
9              [("address", syntax.STRING PhoneBook'x1),
10              ("name", syntax.STRING PhoneBook'x2),
11              ("numbers", syntax.ARRAY (...)),
12              ("type", ...)]])) of
13     (syntax.UNIT ()) => ()
14     | _ => raise Runtime.Exception "Bad return type")
```

Nesting of types can be handled at an arbitrary level as indicated by the previous example. We simply wrap/unwrap each level at a time.

5 Embedding Enumerations, Untyped Values, and Maps

In the previous section, we generated all stubs before CPN Tools got hand on the simulator. This was possible because we had enough type information on the Java side to generate functions that can be used directly from CPN Tools (and which correspond to the simple types, list types and record types). However, some times this may not be enough.

For example, in CP-nets one often use either the enum or index colours. These types are not native to SML, so CPN Tools therefore generates an SML datatype. In Listing 10 we see the essentials of the definitions of the colours A and B in SML.

Now the problem is that two data-types, even if using identical definitions are not considered to be the same type by SML. This will result in a type problem with the code in Listing 11, as the function, test (l. 2) depends on the first definition of MY_TYPE (l. 1), whereas the application of the function (l. 4) uses the second definition (l. 3).

Listing 10 Definition and representation of enum and index data types.

```
1 (* colset A = with red | blue; *)
2 datatype A = red | blue;
3
4 (* colset B = index d with 1..3; *)
5 datatype B = d of int;
```

Listing 11 This code will fail in SML.

```
1 datatype MY_TYPE = A | B
2 fun test A = true | test B = false
3 datatype MY_TYPE = A | B
4 test A
```

Thus we cannot just emit sensible types and hope that they will work, as CPN Tools will override these, and we will get type problems exactly like the ones in Listing 11.

The rest of this section will deal with how we can circumvent this problem. Sect. 5.1 shows how we can use Java enumerations in SML as if they were native types. Then we will show a general solution to the overriding problem described above in Sect. 5.2. Finally we will demonstrate how the general solution can be used to create animation objects that can be used with more than one colour-type.

5.1 Enumerations

Starting with Java version 5.0, Java has native support for enumerations. Enumerations are in principle just standard classes, but they have a simpler syntax. In previous versions of Java, enumerations was often simulated by creating integer constants with the desired names.

We will use a translation from the modern concept to the old concept, and represent functions using enumerations as functions using integers. In Java this can be done by using simple conversion functions. Assume we have defined an enumeration PersonType like in Listing 12 on line 1. We can then use the ordinal() method, defined on any enumeration-type, directly, to convert it to an integer (ll. 1–3). To convert an integer back to the correct enumeration type (ll. 5–10), we need to go through a bit more work, where we basically get all enumeration values in order (l. 6) and return the one, which occurs at the correct position (l. 9). If the index is outside the bounds of the array, we throw an exception (ll. 7–8).

Using reflection, we can write a generic version of the above code; the toInt method is easy as all enumerations inherit from the abstract type java.lang.Enum, which has an ordinal method. We then obtain the method in lines 1–3 in Listing 13. Again, the other direction is more complex, as we have to use reflection. We have extended the method to take a Class parameter, which is the description of the type (l. 5). We now get a description of the values method (l. 8), invoke

Listing 12 Example of how to convert enumerations in Java to and from integers.

```
1  int toInt(PersonType personType) {
2      return personType.ordinal();
3  }
4
5  PersonType fromInt(int number) throws IllegalArgumentException {
6      PersonType array[] = PersonType.values();
7      if (number < 0 || number >= array.length)
8          throw new IllegalArgumentException();
9      return array[number];
10 }
```

the method to get the array of values (l. 9), and finally index into the array using reflection (l. 10). If anything goes wrong (e.g. the type parameter is not an enumeration, the index is wrong, etc.) we catch the error (l. 11–12) and throw an exception (l. 13).

Listing 13 Code to convert generic enumerations in Java to and from integers.

```
1  int toInt(Enum value) {
2      return value.ordinal();
3  }
4
5  Enum fromInt(Class<? extends Enum> clazz, int number)
6      throws IllegalArgumentException {
7      try {
8          Method m = clazz.getDeclaredMethod("values");
9          Object array = m.invoke();
10         return Array.get(array, number);
11     } catch (Exception e) {}
12     throw new IllegalArgumentException();
13 }
```

Using the above code, we can write a generic invocation-handler, which can be used in place of the real animation plug-in. The invocation-handler will pass on any method-calls, but will translate any enumeration parameter/return-value from/to integers. We can thus embed Java enumeration types into CPN-ML.

5.2 On-the-fly Generation of Animation Stubs

Now, we have eliminated the use of enumerations and converted them into a more generic type, which comprises all enumeration types. This will cause us to lose some type-safety. By the translation from the previous section, these methods can be regarded as taking and returning integers.

Now, if we have defined the colour PERSON_TYPE in CPN-ML as in Listing 14 (l. 1), we can use the new addPerson, with the extended Person type either as in Listing 14 lines 3–6, by using the enum type from CPN-ML, but it is also possible to use it as in lines 7–10, using integers directly. This second option has at least two problems. Firstly, it is easily possible to use another value for the type than the legal values, 0, 1, and 2, and secondly, if we had written the enumeration values in another order, e.g. colset PERSON_TYPE = with Child | Male | Female, suddenly Male in CPN Tools would correspond to Female in Java, which is clearly undesirable.

Listing 14 Definition of a CPN-ML type corresponding to the PersonType and use of the type.

```
1  colset PERSON_TYPE = with Male | Female | Child;
2  structure phonebook = PhoneBook(val name = "Phone Book");
3  val _ = phonebook.addPerson { name = "Michael",
4                                address = "Aarhus",
5                                numbers = [1234],
6                                type = PERSON_TYPE.ord Male}
7  val _ = phonebook.addPerson { name = "Kristian",
8                                address = "Aarhus",
9                                numbers = [5678],
10                               type = 0}
```

To overcome these problems, we need to generate type-safe versions of these functions. We do this by extending the way code is injected into the simulator slightly. In Fig. 3 we see a refined view of the initialisation from Fig. 2. The first half is almost the same as Fig. 2, except we now remember any plug-in with unsafe methods (such as enumerations). Now, whenever we check a type-declaration, we forward the check to the simulator, and check if it was valid. This is possible as we have not handled over the control of the simulator directly, but mediated all calls to the simulator and can inspect what is sent back and forth. If the newly defined colour can help us make another plug-in type safe, we generate a type-safe version of it and injects this into the simulator. After that we return the result of the declaration check to the CPN Tools GUI.

For enumeration types we can use the ord and col functions defined by the simulator. These functions will convert a CPN-ML enumeration type to and from integers. We can check that names of the types and enumeration values matches[1]. If we need to do reordering (if the values in Java and CPN-ML are not in the same order), we can easily generate our own translation functions. Listing 15 shows an example where reordering is necessary for type safety.

[1] E.g. PersonType matches PERSON_TYPE because if we disregard hyphenations and underscores and translates the names to all lowercase, they are the same, and if we do the same with the enumerated values, they also correspond to each other

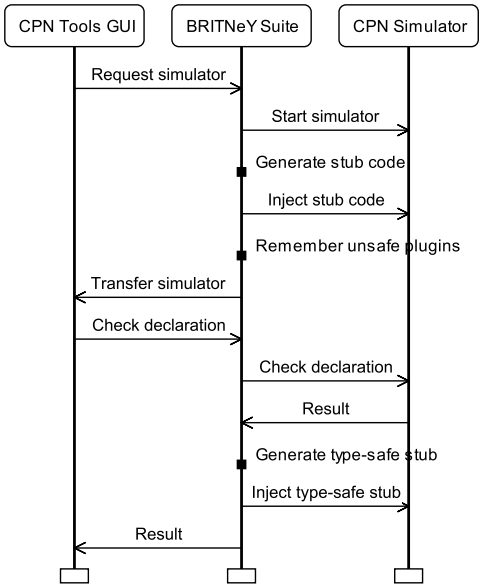

Fig. 3. Extended view of how BRITNeY Suite injects code into the simulator in order to allow the generation of type-safe functions.

Listing 15 Incompatible enumeration colour-set and Java enumeration.

```
1 (* Order of CPN-ML enumartion definition: *)
2 (*           Male: 0, Female: 1, Child: 2 *)
3 colset PERSON_TYPE = with Male | Female | Child;
4
5 // Order of Java enumeration definition:
6 //          Male: 0, Female: 1, Child: 2
7 public enum PersonType { Child, Female, Male };
```

The type-safe stub will then translate the parameters from CPN-ML into unsafe integers and use the unsafe version of the plug-in, injected before the control of the simulator was transferred.

5.3 Untyped Values and Maps

The strategy used in the previous section, where we generate type-safe versions of the stubs lazily, can also be used to simulate a type hierarchy in SML.

In Java it is possible to have a method, which takes as parameter an Object, line addNote on line 6 in PhoneBookInterface from Listing 1. This method can be used to add notes of any kind, e.g. addNote(1234, "PhD student"), addNote(1234, person), and addNote(1234, 5678). It would be nice to be able to do thing like

this in SML, but unfortunately SML has no hierarchy of types, so this is not immediately possible.

We can use the above lazy generation, however. We simply schedule any plug-in with methods that take either parameters or return values that are more generic than we can handle for lazy stub generation. Then, when we encounter a type declaration, we check if we can generate a type-safe version of the stub. In fact, more than one colour can match each method, say we have declared the colours STRING and INT in CPN Tools as strings and integers respectively. Both of these values applies to our addNote method. We solve this problem by generating a function for each matching type, i.e. addNote'STRING: string –>unit and addNote'INT: int –>unit. These will be converted to the correct Java type automatically, and passed as parameters. We shall allow such function generations for any CPN-ML type, that is a sub-type of the corresponding Java-type. In Fig. 4 we see some the Java's type hierarchy. Under each type we have indicated which CPN-ML types it corresponds to (if any). Types in parentheses is not completely supported by CPN Tools currently. Each type has an arrow pointing to it's super-type. For each Java type, we will generate type-safe versions of all sub-types.

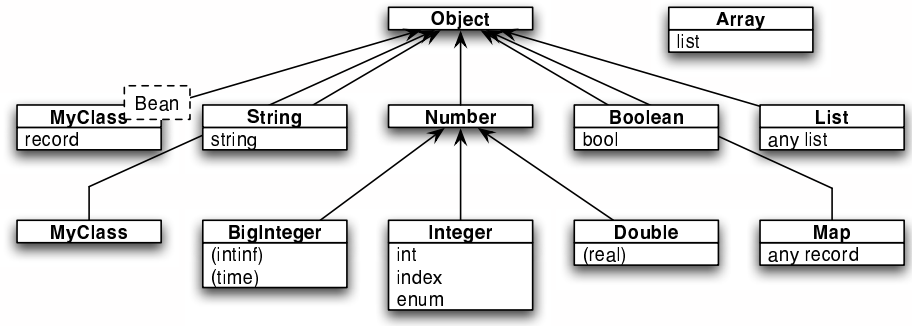

Fig. 4. Some selected Java types.

One special case to take note of, is that all CPN-ML record types are sub-types of the Java type Map. Map is a generic mapping, but without more information, we cannot generate a stub for a map, as we have no information about the key type or about the value types. If we generate a typed variant for each kind of record, however, we get enough information from the colour declaration to generate sensible functions.

If, for instance, we generate the function getName as in Listing 16. This method disregards all other fields and also the actual type of the name.

If we have some colour definitions, as in Listing 17. We would then get functions for each of these four colour-definitions, they would be typed to take exactly a record of the correct type, and they would return a string representation of

Listing 16 Method for returning the name in any record.

```
1 public String getName(Map record) {
2    Object name = record.get("name");
3    return name.toString();
4 }
```

the name, much like in Listing 18, where we see the result of using the function for all four record types. Of course, we would get a type error, should we try something like getName'CONTACT {name = "Michael", age = 25}.

Listing 17 Various record definitions.

```
1 colset PERSON = record name : string * age : int;
2 colset CITY = record name : string * country : string;
3 colset NAMES = record given : string * family : string;
4 colset CONTACT = record name : NAMED * phone : int;
```

Listing 18 Example of use of functions taking arbitrary record.

```
1  getName'PERSON {name = "Michael", age = 25}
2  (* Result: "Michael" *)
3  getName'CITY {name = "Aarhus", country = "Denmark"}
4  (* Result: "Aarhus" *)
5  getName'NAMES {given = "Michael", family = "Westergaard"}
6  (* Result: Exception, no name field *)
7  getName'CONTACT {name = {given = "Michael",
8                           family = "Westergaard"},
9                   phone = 1234}
10 (* Result: "[given => Michael, family => Westergaard]" *)
```

5.4 Complex Example

Finally, we would like to show that it is possible to combine simple types, arrays, Java beans and enumeration in the interface of a Java method and create SML stubs as in Listing 19 (cf. getPersons in Listing 1). The method has the interface PersonInterface[] getPersons(String address).

6 Conclusion

In this paper we have showed how it is possible to embed Java classes in CPN Tools, by describing exactly how Java classes are represented in SML. The current version of BRITNeY Suite implements all the facilities in Sect. 2, 3 and 4. The functionality in Sect. 5 will be added later.

Listing 19 getPersons representation in Java.

```
1  fun getPersons object
2      (PhoneBook'x1) =
3      (case Runtime.invoke(
4          object ^ ".getPersons",
5          [syntax.STRING PhoneBook'x1]) of
6      (syntax.ARRAY PhoneBook'x0) =>
7         List.map
8             (fn (syntax.STRUCT
9                      [("name", syntax.STRING PhoneBook'x2),
10                      ("address", syntax.STRING PhoneBook'x3),
11                      ("numbers", syntax.ARRAY PhoneBook'x4),
12                      ("type", syntax.INT PhoneBook'x5)])
13                 => {name = PhoneBook'x2,
14                     address = PhoneBook'x3,
15                     numbers = List.map
16                                  (fn (syntax.INTEGER PhoneBook'x6)
17                                       => PhoneBook'x6)
18                                   PhoneBook'x4,
19                     type = PERSON_TYPE.col PersonType'x5})
20             PhoneBook'x0)
21     | _ => raise Runtime.Exception "Bad return type")
```

6.1 Future Work

Much work have already been done to incorporate Java in CPN Tools, as described in the previous sections. However, so far we have only described interaction with objects in an RPC fashion. An improvement on this would be to introduce communication to objects using real Remote Method Invocation (RMI) [1, Chap. 5]. This way we would be able to handle any object as object references that could be passed around the net as tokens, similar what can be done in object Petri nets [9]; we would only be able to handle Java objects though, so objects cannot be CP-nets that are implemented in SML.

Now, instead of just ignoring Java methods with unknown types, we will emit a structure describing the methods. For a class Person implementing the PersonInterface from Listing 1, we would emit something like Listing 20. Here we define a signature with functions for each Java method (ll. 6–9) as well as a reference type (l. 2) and a constructor (l. 4). The reference type would be of type string (l. 13) to allow us to use it as a token in CPN Tools.

Then another representation of a PhoneBook class implementing the PhoneBookInterface from Listing 1 would have the signature from Listing 21 rather than the ones from Listings 2 and 3. The difference is that we would have added a reference type (l. 5), which is used as the first parameter (rather than a string, as before). Also, the records previously used to describe Java Beans have been replaced by the reference type Person.instance in addPerson and getPersons (ll. 11–12).

Listing 20 SML code for the class Person.

```
signature PERSON = sig
    type instance

    val create_instance : unit -> instance

    val getName : instance -> string -> unit
    val setName : instance -> unit -> string

    ...
end

structure Person : PERSON = struct
    type instance = string

    ...
end
```

Listing 21 RMI interface for the PhoneBook class.

```
signature PHONEBOOK'RMI =
sig
    exception Exception of string

    type instance

    val create_instance : string -> instance

    val getOwnerName: instance -> int -> string
    val lookupNumbers: instance -> string list * string -> int list
    val addPerson: instance -> Person.instance -> unit
    val getPersons: instance -> string -> Person.instance list
end
```

The reason for not doing this in the first place is that is is closer to Java's object oriented way of working, and thus far away from SML's functional way.

References

1. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Weslay, 3rd edition, 2001.
2. CPN Tools. `www.daimi.au.dk/CPNTools`.
3. JavaBeans 1.01 specification. `java.sun.com/products/javabeans/docs/spec.html`.
4. K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts*. Springer-Verlag, 1992.
5. J.B. Jørgensen and K.B. Lassen. Aligning Work Processes and the Adviser Portal Bank System. In *REBNITA05*, 2005.

6. L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. In *Fifth International Conference on Integrated Formal Methods*, 2005. Accepted for Fifth International Conference on Integrated Formal Methods.
7. R. Milner, R. Harper, and M. Tofte. *The Definition of Standard ML*. MIT Press, 1990.
8. A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proc. of ICATPN 2003*, volume 2679 of *LNCS*, pages 450–462. Springer-Verlag, 2003.
9. R. Valk. Petri Nets as Token Objects - An Introduction to Elementary Object Nets. In *Proc. of ICATPN'98*, volume 1420 of *LNCS*, pages 1–25. Springer-Verlag, 1998.
10. M. Westergaard. BRITNeY Suite website. `wiki.daimi.au.dk/britney/`.
11. M. Westergaard and K.B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ICATPN 2006*, volume 4024 of *LNCS*, pages 431–440. Springer-Verlag, 2006.