Preserving Architectural Properties in Multithreaded Code Generation

Marco Bernardo and Edoardo Bontà

Università di Urbino "Carlo Bo"
Istituto di Scienze e Tecnologie dell'Informazione
Piazza della Repubblica 13, 61029 Urbino, Italy
{bernardo, bonta}@sti.uniurb.it

Abstract. Architectural descriptions can provide support for a formal representation of the structure and the overall behavior of software systems, which is suitable for an early assessment of the system properties as well as for the automated generation of code. The problem addressed in this paper is to what extent the properties verified at the architectural level can be preserved during the code generation process for multithreaded programs. In order to limit the human intervention, we propose to separate the thread synchronization management from the thread behavior translation. While a completely automated and architecture-driven approach can guarantee the correct thread coordination, we show that only a partial translation based on stubs is possible for the behavior of the threads, with the preservation of the architectural properties depending on the way in which the stubs are filled in.

1 Introduction

One of the major objectives of the software architecture level of design [14, 15] is that of producing a reference document – shared by all the people involved in the development process – which describes the structure of the software system as well as the main functional and non-functional aspects of its overall behavior. Whenever such a document is made formal through the use of a suitable architectural description language (ADL), an early assessment of the gross system properties can be carried out. This is the case with process algebraic ADLs, for which several techniques based on equivalence checking have been developed for the component-oriented verification and diagnosis of architectural mismatch freedom [3, 11, 10, 9, 6, 5, 1].

As observed in [7], one of the big issues in the software engineering field is guaranteeing that the implementation of a software system conforms to its architectural description. In other words, a way has to be found to check whether the properties verified at the architectural level are preserved at the code level. In this respect, it may be helpful to generate code directly from the architectural description, as the latter represents an abstract model of the final system. Indeed, the purpose of automatic code generation should be not only to speed up the system implementation, but also to ensure conformance by construction.

In order to reconcile the (architecture-based) early property assessment and the (architecture-driven) automated code generation, in this paper we investigate to what extent the architectural properties can be preserved during the translation of architectural descriptions into code. On the upstream side we shall concentrate on process algebraic architectural descriptions, while on the downstream side we shall focus on multithreaded Java programs.

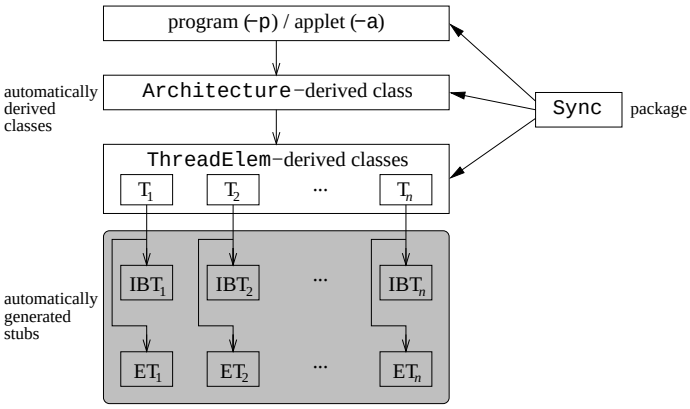

Fig. 1. File structure of the generated code

As discussed in [4] and recalled in Sect. 2, a completely automated and architecture-driven approach can guarantee the correct thread coordination. Given an architectural description in PADL [1], the approach of [4] is based on the use of a translator, called PADL2Java, and a Java package, called `Sync`. The package is structured around four conceptual layers: `Architecture`, `ThreadElem`, `Port`, and `Connector`. As shown in the upper part of Fig. 1, the translator synthesizes a class derived from `Architecture` plus as many classes derived from `ThreadElem` as there are architectural element types in the PADL description. The instances generated for the `ThreadElem`-derived classes, i.e. the threads, are guaranteed to interact as expected thanks to the generation of the appropriate instances of `Port` and `Connector`. On top of the previous classes, there may be another class with the `main()` method or an `Applet`-derived class, depending on the option with which PADL2Java is invoked.

In order to limit the human intervention, in this paper we propose to separate the thread synchronization management from the thread behavior handling, then we complete the approach of [4] by taking care of the thread behavior translation. In the process algebraic framework of PADL, the behavior of an architectural element – to be implemented as a thread – is given through a sequence of equations based on a restricted number of operators: action prefix, choice, and behavior invocation. In Sect. 3 we show how such process algebraic operators can be translated into the Java framework. In particular, for the action prefix we shall see that a different treatment is needed depending on whether the action

– which corresponds to a sequence of thread statements – is an internal action or an interaction. In fact, while the latter is involved in the communications, hence it is managed as explained in [4], the former can only be rendered as a stub during the translation of the thread behavior. More precisely, we shall have stubs for the actions internal to the behavior of the threads (IBT), together with stubs for the handling of the exceptions raised by the interactions of the threads (ET), as shown in the lower part of Fig. 1. This means that the translation of the thread behavior cannot be completely automatic, as should be expected due to the different levels of abstraction of an ADL and a programming language.

On the theoretical side, we show in Sect. 4 that the preservation of the architectural properties during the translation process critically depends on the way in which the IBT and ET stubs are filled in. We shall provide to this purpose a set of guidelines that guarantee the property preservation along the translation.

In Sect. 5 some remarks are finally reported about related and future work.

2 Thread Coordination

In this section we recall the approach proposed in [4] to ensure a correct thread coordination for Java programs generated from PADL descriptions.

2.1 The Language PADL

Every PADL description comprises two sections. In the first section, the types of architectural elements of the software system are defined by specifying their behavior and their interfaces. The behavior of an architectural element type is described through a sequence of behavioral equations of the form $B(\underline{x}) = P$. The process algebraic term P has the following syntax:

$$P ::= \texttt{stop} \mid \texttt{a}.P' \mid \texttt{choice}\{P, \ldots, P\}$$
$$P' ::= B(\underline{e}) \mid P$$

where `stop` is the terminated process, $\texttt{a}.P$ is an application of the action prefix operator, $\texttt{choice}\{P_1, ..., P_n\}$ is an application of the choice operator, and $B(\underline{e})$ is a behavioral equation invocation.

The interfaces are actions occurring in the behavioral equations, which are used to interact with other architectural elements in the system. Each interaction has three qualifiers associated with it. The first qualifier establishes whether the interaction is an input (receiver side) or an output (sender side). The second qualifier determines the multiplicity of the communications in which the interaction can be involved: one-to-one (uni-interaction), conjunctive one-to-many (and-interaction – e.g. broadcast), and disjunctive one-to-many (or-interaction – e.g. client-server). The third qualifier establishes whether the interaction participates in a communication in a synchronous or asynchronous way.

In the second section of a PADL description, the topology of the system is defined by declaring the instances of the previously defined types of architectural elements, the interactions that act as interfaces for the whole system (useful for hierarchical modeling), and the attachments among the other interactions of the architectural element instances.

We conclude by exemplifying PADL through the description of a simple architectural element type, which will be reused in Sect. 3.6 to illustrate the thread generation. The architectural element `Checker_Type` that we consider gets values from another architectural element until a value is received that coincides with an expected one, then forwards the value to another architectural element and terminates. The values that are different from the expected one are printed out as long as they are received. Here is the PADL description of `Checker_Type`:

```
ELEM_TYPE Checker_Type(const integer expected_value)
  BEHAVIOR
    Getting_Value(void; local integer received_value) =
      <get_value?(received_value), _> . Checking_Value(received_value);
    Checking_Value(integer received_value; void) =
      choice {
        cond(received_value = expected_value) ->
          <prepare_to_forward, _> .
            <forward_value!(received_value), _> . stop,
        cond(received_value != expected_value) ->
          <prepare_to_print, _> . Printing_Value(received_value)
      };
    Printing_Value(integer received_value; void) =
      <print_value!(received_value), _> . Getting_Value()
  INPUT_INTERACTIONS UNI SYNC get_value
  OUTPUT_INTERACTIONS UNI SYNC forward_value
```

`Checker_Type` has three behavioral equations: `Getting_Value`, `Checking_Value`, and `Printing_Value`. Within the equations, every action is represented through its name – possibly followed by a parameter that is separated from the action name by "`?`" (resp. "`!`") in the case of an input (resp. output) action – and the information about its priority level and its weight – "`_`" is used to denote the default values. Unlike the other actions, the two actions `prepare_to_forward` and `prepare_to_print` are preceded by a boolean guard, which establishes the condition under which these actions can be executed based on the comparison of the received value with the expected one. These two actions together with `print_value` are internal, whereas `get_value` is an input synchronous uni-interaction and `forward_value` is an output synchronous uni-interaction.

2.2 The Java Package `Sync`

The package `Sync` offers a set of facilities to support the development of multi-threaded Java programs by handling the details of the thread synchronization in a way that is transparent to the software developer. This package is organized into four conceptual layers: `Architecture`, `ThreadElem`, `Port`, and `Connector`. Each layer corresponds to a different architectural abstraction and comprises a set of components realized through Java classes and interfaces.

The first layer, `Architecture`, is an abstract class containing components belonging to lower layers. On the implementation side, `Architecture` is derived from the class `ThreadElem`. This means that an arbitrarily complex architecture is a thread, which is useful for systems modeled in PADL in a hierarchical way.

The second layer, `ThreadElem`, is a class that inherits from the Java class `Thread`. The reason why a derived class has been defined instead of directly using the class `Thread` is related to the translation of the behavior of the architectural elements of a PADL description into the corresponding Java threads. As we shall see in Sect. 3, this is accomplished through additional methods made available in the class `ThreadElem`.

The third layer, `Port`, realizes the abstraction corresponding to a set of statements through which a thread interacts with other threads. Based on the synchronization model adopted in PADL, there are twelve types of `Port`: six are synchronous and six are asynchronous. If a `Port` is synchronous, it waits – and the thread that contains it passivates – until the communication has been established. If a `Port` is asynchronous, it communicates if the other connected `Port` is willing to communicate with it, otherwise an exception is raised and no communication takes place. Both in the synchronous and in the asynchronous mode, three types of `Port` are on the sender side and the other three are on the receiver side. A sending `Port` can transfer to one or more receiving `Port`s an array of generic `Object`s using the method `send()`. Similarly, a receiving `Port` can receive from one or more sending `Port`s an array of generic `Object`s using the method `receive()`. A null array represents a pure synchronization signal. Both on the sender and on the receiver side, the three types are termed "uni", "and", and "or". A uni-`Port` interacts with only one `Port` of another thread. An and-`Port`, instead, interacts with several `Port`s of other threads in a broadcast fashion. Finally, an or-`Port` interacts with only one `Port` of a thread selected out of a set of other threads.

The fourth layer, `Connector`, sets up a communication link between a sending `Port` and a receiving `Port` of two different threads. There are four types of `Connector` in order to cover all the possible combinations of the types (synchronous vs. asynchronous) of the two connected `Port`s. The package contains an internal mechanism – through the method `attach()` defined in the layer `Architecture` – that, given two `Port`s, creates a `Connector` of the right type.

2.3 The Translator PADL2Java

Once the PADL description of a multithreaded Java program has been provided, the software developer can use the translator PADL2Java to generate a skeleton of the program itself. As illustrated in the upper part of Fig. 1, the automatically generated Java code imports the package `Sync` and is composed of several classes derived from `ThreadElem`, each representing the corresponding architectural element type defined in the PADL description, as well as a class derived from `Architecture`, which corresponds to the overall PADL description.

The `Architecture`-derived class has five completely specified sections: *declaring threads*, *declaring architectural interactions*, *defining constructor*, *building architecture*, and *running architecture*.

Each `ThreadElem`-derived class has instead the following four sections: *defining constructor*, *defining behavior*, *instantiating input interactions*, and *instantiating output interactions*. All of these sections are completely specified, except

for *defining behavior*, whose generation is the focus of this paper and will be discussed in Sect. 3.

Finally, in the automatically generated code there is in addition a class with the `main()` method or an `Applet`-derived class, whenever PADL2Java is invoked with option `-p` (for program) or `-a` (for applet), respectively. If option `-c` (for class) is used, then no wrapper class is added to the generated code.

3 Thread Behavior

The approach of [4] only deals with thread coordination. The contribution of this section is to show how to translate the PADL description of an architectural element type into a thread. The resulting code will fill in the *defining behavior* section of the corresponding `ThreadElem`-derived class, as anticipated in Sect. 2.3.

The basic idea is that, besides the method `run()` already defined as abstract in the Java base class `Thread`, within the *defining behavior* section we need to generate some additional methods that are the Java translation of the PADL behavioral equations. The generation of these behavioral methods must then be complemented by the generation of some stubs, as depicted in the lower part of Fig. 1. Each such stub is a class containing either a method for every internal action occurring in the PADL description of the behavior of the thread (IBT), or a method for every interaction occurring in the PADL description of the behavior of the thread that can result in an exception to be handled (ET).

Thus, for every architectural element type contained in a PADL description, PADL2Java will have to generate not only a `ThreadElem`-derived class, but also two classes for the internal action translation and the interaction-related exception handling, respectively. The `ThreadElem`-derived class will declare as protected members two objects of class IBT and ET, respectively, and will instantiate these objects within the method `run()`.

Unlike the method `run()` and the behavioral methods, which are completely generated in an automatic way, the methods contained in the IBT and ET stubs have to be manually filled in by the developer. The reason is that, in the case of the IBT stubs, the methods are associated with the execution of the internal actions. An internal action describes at a high level of abstraction a set of operations to be carried out by the thread, so in general it will correspond to a sequence of Java statements to be defined by the developer. Likewise, the way in which an exception raised by an interaction has to be handled must be established by the developer.

In this section, we first analyze the execution flow of a thread and show how to generate the method `run()` avoiding recursion (Sect. 3.1). Then we show how to generate the code for the behavioral methods by proceeding by induction on the syntactical structure of the process algebraic terms occurring in the right-hand side of the corresponding behavioral equations (Sect. 3.2 to 3.5). Finally, we show an example of Java thread code generated from the PADL description of a specific architectural element type (Sect. 3.6).

3.1 The Execution Flow and the Method `run()`

The execution flow of a Java thread generated with PADL2Java is determined by the behavioral equations – translated into as many behavioral methods – of the corresponding architectural element type of the PADL description. In order to generate efficient Java code, in the translation process we have to get rid of the frequently occurring recursive behavioral invocations. In the process algebraic syntax only tail recursion comes into play, which is easy to transform into iteration.

Following [8], one possibility is to generate a `while` statement containing an `if-elseif` statement – or equivalently a `switch-case` statement – through which the next behavioral method to be executed is selected based on a label variable. Instead of invoking the next method, before returning each method properly sets the label variable.

Unfortunately this approach is not efficient when several method calls are contained within the conditional statement, because several checks must be done for different cases in every cycle. We therefore propose a variant of this approach, in which the address of the method to be executed is used, instead of a label variable. Since pointers are not available in Java, we exploit Java reflection in a transparent way to accomplish this task.

In order to implement our proposal, three protected methods are added to the class `ThreadElem`. The first method, `behavEqList()`, accepts as input an array of elements of class `BehavEqId`, each of which is an object containing information – name and formal parameters – about a behavioral method belonging to a `ThreadElem`-derived class. The method `behavEqList()` builds a map of the behavioral methods that translate the behavioral equations in the architectural element type for the `ThreadElem`-derived class. This static map is then used to retrieve a behavioral method, given its index (indices start from `0`).

The second method, `behavEqNext()`, requires to specify the index and the actual parameters of the next behavioral method to be executed. The method `behavEqNext()` returns `false` if the index is equal to or greater than the number of behavioral methods, or if the actual parameters do not match the formal parameters previously specified with `behavEqList()`. Otherwise, the method `behavEqNext()` returns `true` and the method retrieved from the map is placed into a private `ThreadElem` member variable. The value of this variable is set to `null` if the next behavior of the thread is described by process term `stop`.

The third method, `behavEqCall()`, invokes the behavioral method previously placed in the private `ThreadElem` member variable by `behavEqNext()`. The method `behavEqCall()` returns `true` if the last invocation of `behavEqNext()` has been successful and the value of the private `ThreadElem` member variable is not `null`. The returned value determines whether the body of the `while` statement has to be repeated or not.

In the automatically generated code, `behavEqList()` with the appropriate parameters is invoked by the constructor of the `ThreadElem`-derived class. The methods `behavEqNext()` and `behavEqCall()`, instead, are invoked by `run()`.

The latter is redefined in the *defining behavior* section of the `ThreadElem`-derived class, just before the definition of the behavioral methods:

```
public void run() {
  <IBT instantiation>
  <ET  instantiation>
  behavEqNext(0, <actual parameters of the first equation>);
  while (behavEqCall());
}
```

The call to the method `behavEqNext()` specifies the first behavioral method to be executed, which has index `0` in the map, together with its actual parameters.

3.2 Behavioral Invocations

The behavioral invocation $B(\underline{e})$ represents a process term that behaves as the behavioral equation whose identifier is B, when passing the possibly empty sequence of actual parameters $\underline{e}$. A behavioral invocation, which can occur only within the scope of an action prefix operator, is not translated into a behavioral method call, as this may result in the generation of inefficient code in case of recursion. Instead, a behavioral invocation is translated into an invocation of `behavEqNext()`, to which the index and the actual parameters of the behavioral method associated with the invoked behavioral equation are passed.

3.3 Stop

Process term `stop` represents the situation in which no further action can be executed. The process term `stop` is translated into an invocation of the method `behavStop()` defined in the class `ThreadElem`. The method `behavStop()` is similar to the method `behavEqNext()`, but it does not return any value, it has no input parameters, and causes the method `run()` to terminate because the next invocation of `behavEqCall()` returns `false`.

3.4 Action Prefix

The action prefix operator is used to represent a process term that can execute an action and then behaves as described by another process term. Every action has three pieces of information associated with it: (*i*) a boolean guard, expressing the possible constraint under which the action can be enabled (default value `true`), (*ii*) a positive integer number representing a priority level, which is used when resolving choices among several enabled actions (default value `1`), and (*iii*) a positive real number representing a weight, which is used when resolving choices among several enabled actions with the same priority (default value `1.0`).

In PADL an action can be an interaction or an internal action. In the `Sync`-based thread coordination, the output/input interactions are translated into invocations of the methods `send()`/`receive()` within the instances of the classes of layer `Port`, as described in Sect. 2.2. However, since `send()` and `receive()`

can be subject to the following two exceptions, the translation of the interactions must be completed by filling in the corresponding ET stubs.

The first exception, `UnattachedPortException`, is raised when an architectural interaction is executed, which is not attached to any other interaction. In this case, the architectural interaction is like an internal action, hence the sequence of Java statements translating it has to be manually provided.

The second one, `AsyncPortNotReadyException`, is raised when an asynchronous interaction is executed, but the other party is not ready to communicate with it. In this case, the thread containing the asynchronous interaction goes on, but the developer may want to add some Java statements to deal with this event.

Unlike the interactions, which are completely translated in an automatic way up to the handling of the exceptions that they may rise, the internal actions cannot be treated automatically at all. A method for each of them is placed in an IBT stub, which has to be filled in by the developer with the corresponding Java statements. As a consequence, every occurrence of an internal action is translated into an invocation of the related method in an IBT stub.

3.5 Choice

The choice operator expresses a selection among a certain number of alternative behaviors described through process terms. A choice-based process term is translated into a `switch-case` statement, whose condition is given by an invocation of the method `choice()` defined in the class `ThreadElem`.

There are two cases that must be addressed in order to translate the choice operator. The first one is the case where every process term involved in the choice starts with an action prefix operator. In this case the method `choice()` is directly employed, which accepts as input an array of objects of class `ChAct`, each of which contains the three pieces of information mentioned in Sect. 3.4 about one of the starting actions. Should one of the starting actions be an interaction, an additional piece of information is contained in the corresponding object, which is a reference to the object `Port` associated with the interaction. The method `choice()` returns the index (within the array) of the starting action selected for execution.

A starting action can be selected if: (*i*) its guard evaluates to `true`, (*ii*) the corresponding `Port` object is ready to send or receive, whenever the starting action is an interaction, and (*iii*) its priority is not less than the priority of the other enabled starting actions, with its weight being used to probabilistically solve the choice among the enabled starting actions with the highest priority. If all the enabled starting actions are interactions, the method `choice()` waits – and the thread that contains it passivates – until one of the associated `Port`s is ready to communicate. If the array that contains the objects of class `ChAct` is empty or all the guards of the starting actions evaluate to `false`, the method `choice()` returns a negative value.

Based on the index returned by `choice()`, the `switch-case` statement invokes the method associated with the execution of the selected starting action. This method is `send()` or `receive()` in the case of an interaction, whereas for

an internal action it is the corresponding method in an IBT stub. The invocation of this method is followed in turn within the `switch-case` statement by the translation of the process term prefixed by the selected action. In the default clause, which comes into play when a negative value is returned by `choice()`, the method `behavStop()` is invoked.

The second case is the one in which some of the process terms involved in the choice do not start with an action prefix operator. If one of these process terms is `stop`, then nothing has to be added for it in the `ChAct` array and the `switch-case` statement, because the method `behavStop()` is selected by default whenever the other involved process terms cannot be selected. If instead one of these process terms is a nested choice, then a flattening of the nested choice takes place during the translation. This means that the `ChAct` array and the `switch-case` statement for the outer choice are extended in order to include all the alternative starting actions that are contained in the inner choice. The event in which one of the process terms involved in the choice is a behavioral invocation cannot happen, because a behavioral invocation can only occur within an action prefix operator.

3.6 Example of Thread Behavior Generation

In order to illustrate all the features of the proposed approach, we conclude by showing the translation into a Java thread of the architectural element type described with PADL in Sect. 2.1. A more complex and realistic example can be found at `http://www.sti.uniurb.it/bonta/java_audio_proc/`.

Below we exhibit the *defining behavior* section automatically generated for the `ThreadElem`-derived class `Checker_Type`, which comprises the method `run()` along with the three behavioral methods associated with the three behavioral equations of the considered architectural element type:

```
public void run() {
  act_Checker_Type = new IBT_Checker_Type();
  // no ET instantiation as there are
  // no architectural interactions and no asynchronous interactions
  behavEqNext(0, null);
  while (behavEqCall());
}
public void Getting_Value() {
  Integer received_value;
  try {
    Object obj[] = get_value.receive();
    received_value = (Integer)obj[0];
  } catch (SyncException e) {}
  behavEqNext(1, new Object[] {received_value});
}
```

```
public void Checking_Value(Integer received_value) {
  switch(
    choice(new ChAct[] {
      new ChAct(received_value.intValue() == expected_value, 1, 1.0),
      new ChAct(received_value.intValue() != expected_value, 1, 1.0)
    })
  )
  {
    case 0:
      act_Checker_Type.prepare_to_forward();
      try {
        forward_value.send(new Object[] {received_value})
      } catch (SyncException e) {}
      behavStop();
      break;
    case 1:
      act_Checker_Type.prepare_to_print();
      behavEqNext(2, new Object[] {received_value});
      break;
    default:
      behavStop();
      break;
  }
}
public void Printing_Value(Integer received_value) {
  act_Checker_Type.print_value(received_value);
  behavEqNext(0, null);
}
```

The three behavioral methods are numbered `0`, `1`, and `2` in the array built by the method `behavEqList()` invoked from within the constructor of the Java class `Checker_Type`. Therefore, these are the indices that occur above in the invocations of method `behavEqNext()`.

The interactions `get_value` and `forward_value` are translated into invocations of the methods `receive()` and `send()`, respectively, which are defined in the corresponding `Port`s. Since such methods can throw exceptions of class `SyncException` defined in the package `Sync`, their invocation must be controlled using a `try-catch` statement. The class `SyncException` is the superclass of `UnattachedPortException` and `AsyncPortNotReadyException` mentioned in Sect. 3.4, which need to be handled with suitable methods to be manually defined in the ET stub. Since none of `get_value` and `forward_value` is architectural or asynchronous, no exception handler is needed for them, which explains why no ET stub is instantiated by the method `run()`.

The three methods `prepare_to_forward()`, `prepare_to_print()`, and `print_value()` related to the internal actions are invoked on the object `act_Checker_Type`, which is of class `IBT_Checker_Type`. This object is declared as a protected member within the class `Checker_Type` and is then instantiated within the method `run()` above. The class `IBT_Checker_Type` is defined as follows:

```
class IBT_Checker_Type {
  // ADD CLASS MEMBER DECLARATIONS IF NEEDED
  IBT_Checker_Type() {
    // FILL IN THE CONSTRUCTOR BODY IF NEEDED
  }
  void prepare_to_forward() {
    // FILL IN THE METHOD BODY
  }
  void prepare_to_print() {
    // FILL IN THE METHOD BODY
  }
  void print_value(Integer received_value) {
    // FILL IN THE METHOD BODY
  }
}
```

The methods associated with the three internal actions will have to be manually filled in by the developer based on the semantics of the internal actions themselves. The developer is also allowed to fill in the body of the constructor of the class `IBT_Checker_Type` and to add member declarations whenever needed.

4 Preservation of Architectural Properties

PADL is equipped with a component-oriented technique based on equivalence checking for verifying the freedom from architectural mismatches [1]. These are the malfunctionings that arise when assembling together several components that are correct if considered in isolation. More precisely, the class of properties dealt with by the technique – which includes for instance deadlock freedom – is characterized by three constraints. First, the properties can only be concerned with the interactions, as they are the actions through which the components communicate. Second, for each property $\mathcal{P}$ in the class, there must exist a weak equivalence $\approx_{\mathcal{P}}$ coarser than $\approx_{\mathrm{B}}$ (weak bisimulation [13]) that preserves $\mathcal{P}$ – it never equates two process terms such that one of them satisfies $\mathcal{P}$ while the other does not – and is a congruence with respect to the static process algebraic operators. Third, the (action-based) temporal logic in which the properties of the class are expressed cannot allow the negation to be freely used.

An important issue is to guarantee that the properties proved at the architectural level are then preserved at the code level. Since we have taken an approach based on automatic code generation, property preservation should be achieved by construction. In other words, the translation from PADL to Java illustrated before should have been defined in a way that ensures the property preservation. This is what we are going to investigate in this section.

4.1 Code Generated for the Thread Management

The code for handling the threads is completely generated in an automatic way by means of the package `Sync`. As far as the system topology is concerned, this

is built in the `Architecture`-derived class in the same way as prescribed by the second section of the PADL specification.

As far as the thread coordination is concerned, both PADL and `Sync` adhere to the same synchronization model. On the PADL side, each interaction is given three qualifiers: output vs. input, uni vs. and vs. or, synchronous vs. asynchronous. Each interaction is then translated into an invocation of the method `send()` or `receive()` defined in the corresponding `Port`, depending on whether it is an output or an input interaction, respectively. Additionally, the kind of this `Port` (uni vs. and vs. or, synchronous vs. asynchronous) is the same as that of the interaction.

As a consequence, the code generated for managing the threads cannot infringe the preservation of the architectural properties, up to the methods for handling the exceptions raised by architectural and asynchronous interactions.

4.2 Code Generated for the Behavioral Equations

Each behavioral equation occurring in the PADL description of an architectural element type is translated into a behavioral method of the corresponding `ThreadElem`-derived class. The translation proceeds by induction on the syntactical structure of the process term on the right-hand side of the behavioral equation, based on the operators that occur in such a process term. The way in which the translation is carried out, together with the way in which the thread execution flow proceeds according to the order established by the invocations of the behavioral equations, ensures the preservation of the process algebraic semantics, up to the methods related to the internal actions.

4.3 Code Provided for Filling in the Stubs

In conclusion, the preservation of the architectural properties critically depends on the way in which the developer manually fills in the IBT and ET stubs. Here we shall consider only the IBT stubs, as the ET stubs can be treated similarly.

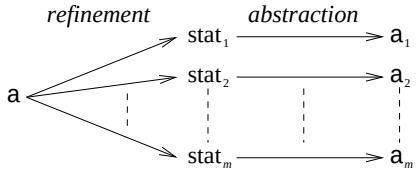

Fig. 2. Internal action refinement and related statement abstraction

In order to be able to reason about the architectural property preservation, we have to compare the internal actions and the corresponding sequences of Java statements on the same process algebraic ground. As shown in Fig. 2, the Java statements into which an internal action is refined during the translation

process can be abstractly viewed as fresh actions. The following theorem provides a sufficient condition for ensuring the preservation of an architectural property of the considered class. Below, we denote by $\_/\_$ the hiding operator and by $\simeq_{\mathrm{B}}$ the observational congruence of [13].

Theorem 1. *Let* T *be the process algebraic description of the behavior of a thread and let* a *be an internal action occurring in* T*. Let* $\mathtt{a}_1$, $\mathtt{a}_2$, ..., $\mathtt{a}_m$ *be the fresh actions abstracting the statements into which* a *is translated and let* T' *be the process algebraic description of the behavior of the thread obtained from* T *by replacing every occurrence of* $\mathtt{a}.\_$ *with* $\mathtt{a}_1.\mathtt{a}_2.\ \ldots\ .\mathtt{a}_m.\_$*. Let* H *be the set of internal actions occurring in* T *or* T'*. Whenever* T *satisfies* $\mathcal{P}$ *and* $\mathtt{a.stop}\ /\ \mathtt{H} \simeq_{\mathrm{B}} \mathtt{a}_1.\mathtt{a}_2.\ \ldots\ .\mathtt{a}_m.\mathtt{stop}\ /\ \mathtt{H}$*, then* T' *satisfies* $\mathcal{P}$ *as well.*

Proof. Since $\simeq_{\mathrm{B}}$ *is a congruence with respect to all the process algebraic operators, from* $\mathtt{a.stop}\ /\ \mathtt{H} \simeq_{\mathrm{B}} \mathtt{a}_1.\mathtt{a}_2.\ \ldots\ .\mathtt{a}_m.\mathtt{stop}\ /\ \mathtt{H}$ *it follows that* $\mathtt{T}\ /\ \mathtt{H} \simeq_{\mathrm{B}} \mathtt{T'}\ /\ \mathtt{H}$*, hence* $\mathtt{T}\ /\ \mathtt{H} \approx_{\mathrm{B}} \mathtt{T'}\ /\ \mathtt{H}$*. Since* $\mathcal{P}$ *must be equipped with a weak equivalence* $\approx_{\mathcal{P}}$ *coarser than* $\approx_{\mathrm{B}}$*, it follows that* $\mathtt{T}\ /\ \mathtt{H} \approx_{\mathcal{P}} \mathtt{T'}\ /\ \mathtt{H}$*. Since* T *satisfies* $\mathcal{P}$*,* $\approx_{\mathcal{P}}$ *preserves* $\mathcal{P}$*, and* $\mathcal{P}$ *can only make assertions about the interactions (which do not belong to* H*), it follows that* T' *satisfies* $\mathcal{P}$ *as well.*

Note that in the theorem above it is not necessarily the case that all of the actions $\mathtt{a}_1$, $\mathtt{a}_2$, ..., $\mathtt{a}_m$ associated with the Java statements provided by the software developer belong to H. As an example, one of such actions may correspond to an invocation of `send()`/`receive()` or of a behavioral method. Fortunately, both cases are prevented from occurring by the fact that the `Port` instances – which contain methods `send()` and `receive()` – and the `ThreadElem`-derived class instances – which contain the behavioral methods – are not visible within the stubs.

We conclude by providing some guidelines that the developer should follow when filling in the stubs in order to preserve the architectural properties:

- No `synchronized` methods should be defined within the stubs, so that methods like `wait()` and `notify()` – which could not be abstracted through internal actions – cannot occur within the stubs.
- No further thread should be created within the stubs, as this would have an observable impact on the system topology and the thread coordination.
- There should be no variables/objects that are visible from several stub classes. This means that all the data shared by several threads should be exchanged only through suitable components of the package `Sync`.
- In the stub method associated with the first internal action following an invocation of the method `send()` (resp. `receive()`), every object that has been passed in that invocation should be copied, with all the stub methods associated with the subsequent internal actions working on that copy of the object. This avoids interferences among threads stemming from the fact that the method `send()` always keeps a reference to the passed objects – so that it can be defined in the package `Sync` in a way that supports arbitrarily many parameters of arbitrary types – and such objects may be modified by the stub method associated with some internal action.

– All the exceptions that can be raised when executing a stub method should be caught or prevented from being raised inside the stub method itself.
– Non-terminating statements should be avoided within the stub methods.

5 Conclusion

In this paper we have addressed the problem of automatically generating multi-threaded programs from formal architectural descriptions, in a way that builds on [4] and preserves the properties proved at the architectural design level. Since the preservation of the architectural properties critically depends on the way in which certain methods are manually filled in by the developer, we have provided some guidelines that should be followed when completing such methods.

Concerning related work that addresses both architecture-driven code generation and architectural property preservation, we have ArchJava and C2SADEL. ArchJava [2] is an extension of Java aiming at the unification of software architecture with implementation, in order to ensure that the implementation conforms to the architectural description with respect to communication integrity. According to this property, each component in the implementation may only communicate directly with the components to which it is connected in the architecture. Our approach differs from ArchJava in several ways. First, it does not extend Java, but generates Java code from process algebraic architectural descriptions. In our approach the developer is then required to fill in some stubs to complete the code for the behavior of the threads, thus giving a certain degree of flexibility. The price to be paid is that the guidelines may be violated, whereas a similar situation is not possible in ArchJava. Second, our approach focuses on the issue of correct thread coordination with respect to a rich synchronization model implemented in `Sync`. This guarantees a property that is even stronger than communication integrity: Implementation threads directly communicate only with the threads they are connected to in the architectural description, in the way prescribed by the architectural description itself with respect to the communication mode (synchronous, asynchronous, asymmetric) and the communication multiplicity (uni-uni, and-uni, or-uni). Third, our approach is more general in the sense that it considers the preservation of a class of architectural properties related to the system behavior, rather than a specific static property. This is formalized through a theorem and a set of guidelines that the developer should follow when filling in the stubs.

C2SADEL [12] is an ADL tied to the C2 style, which combines the usual architectural concepts with type theory. Type checking is used to analyze the architectural descriptions for consistency by unifying corresponding operations required and provided by different components. Moreover, Java code can be automatically generated from C2SADEL descriptions. Since type checking is a static analysis technique, while the architectural properties on which we focus are dynamic and concerned with a rich synchronization model, the differences between our approach and C2SADEL are similar to those above between our approach and ArchJava.

For the future we plan to make some experiments to assess on the field the effectiveness of the proposed framework as well as the performance of the generated code. We also would like to investigate the applicability of our approach to C2SADEL, in order to take advantage of both type checking and behavioral analysis from the architectural level to the code level. Moreover, it would be interesting to combine our approach with ArchJava – by generating ArchJava code instead of Java code – in order to exploit their complementary strengths.

References

1. A. Aldini and M. Bernardo, *"On the Usability of Process Algebra: An Architectural View"*, to appear in Theoretical Computer Science.
2. J. Aldrich, C. Chambers, and D. Notkin, *"ArchJava: Connecting Software Architecture to Implementation"*, in Proc. of the *24th Int. Conf. on Software Engineering (ICSE 2002)*, IEEE-CS Press, pp. 187-197, Orlando (FL), 2002.
3. R. Allen and D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM Trans. on Software Engineering and Methodology 6:213-249, 1997.
4. M. Bernardo and E. Bontà, *"Generating Well-Synchronized Multithreaded Programs from Software Architecture Descriptions"*, in Proc. of the *4th Working IEEE/IFIP Conf. on Software Architecture (WICSA 2004)*, IEEE-CS Press, pp. 167-176, Oslo (Norway), 2004.
5. M. Bernardo, P. Ciancarini, and L. Donatiello, *"Architecting Families of Software Systems with Process Algebras"*, in ACM Trans. on Software Engineering and Methodology 11:386-426, 2002.
6. C. Canal, E. Pimentel, and J.M. Troya, *"Compatibility and Inheritance in Software Architectures"*, in Science of Computer Programming 41:105-138, 2001.
7. D. Garlan, *"Formal Modeling and Analysis of Software Architectures: Components, Connectors, and Events"*, in *Formal Methods for Software Architectures*, LNCS 2804:1-24, 2003.
8. R. Guimarães and W. Borelli, *"An Automatic Java Code Generation Tool for Telecom Distributed Systems"*, in Proc. of the *Int. Conf. on Software, Telecommunications and Computer Networks (SOFTCOM 2002)*, Split (Croatia), 2002.
9. P. Inverardi and S. Uchitel, *"Proving Deadlock Freedom in Component-Based Programming"*, in Proc. of the *4th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2001)*, LNCS 2029:60-75, Genova (Italy), 2001.
10. P. Inverardi, A.L. Wolf, and D. Yankelevich, *"Static Checking of System Behaviors Using Derived Component Assumptions"*, in ACM Trans. on Software Engineering and Methodology 9:239-272, 2000.
11. J. Magee and J. Kramer, *"Concurrency: State Models & Java Programs"*, Wiley, 1999.
12. N. Medvidovic, D.S. Rosenblum, and R.N. Taylor, *"A Language and Environment for Architecture-Based Software Development and Evolution"*, in Proc. of the *21st Int. Conf. on Software Engineering (ICSE 1999)*, IEEE-CS Press, pp. 44-53, Los Angeles (CA), 1999.
13. R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989.
14. D.E. Perry and A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992.
15. M. Shaw and D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996.