Generating Well-Synchronized Multithreaded Programs from Software Architecture Descriptions

Marco Bernardo and Edoardo Bontà
Università di Urbino "Carlo Bo"
Istituto di Scienze e Tecnologie dell'Informazione
Piazza della Repubblica 13, 61029 Urbino, Italy

Abstract

Multithreading provides an adequate support for concurrent programming, but requires the software developer to take care of the correct synchronization and exchange of data among threads. In this paper we propose an architecture-driven approach to the thread synchronization management, which is completely transparent to the software developer. This is realized by implementing a suitable Java package – which adheres to a general synchronization model and is inspired by the main architectural abstractions – by means of which well-synchronized multithreaded Java programs can be synthesized from their architectural specifications. The approach is illustrated by means of a real-time audio processing system.

1. Introduction

The increasing importance of multimedia and real-time applications causes many software developers to often face concurrent programming. One of the most powerful tools to support concurrent programming is multithreading. Unlike traditional concurrent programming, where each code unit is a distinct program executed as a distinct process, multithreading is a technique based on the simultaneous execution of several parts of code – often sharing some data – within a single program. The benefit of this technique is evident in multiprocessor systems, where the concurrency between quasi-independent threads allows the running time to be reduced proportionally to the number of processors. However, even in uniprocessor systems, where parallelism is not real but simulated by means of time sharing, there are several advantages when employing multithreading. First, the management of the system resources is more efficient, because the idle times can be reduced by suitably switching the CPU among CPU-bound and I/O-bound threads. Second, the interaction between the program and its user is fully supported, in such a way that no execution of previously started threads must be interrupted or completed before the user needs to interact with the program again. Third, the exchange of structured data among different threads is made easier by the use of shared memory and shared variables within the same program, without resorting to the more complicated data passing techniques typical of multiprocess software systems.

Multithreading offers a good level of flexibility but, on the other hand, more attention must be paid to the thread synchronization as this is completely entrusted to the software developer. As a consequence, the software developer should be provided with a suitable support for being confident in the correctness of the way in which the thread synchronization is dealt with. This is especially important when developing efficient concurrent programs, as this usually requires a complicated combination of several different synchronization techniques, like e.g. sleep and wakeup primitives, semaphores, and monitors.

The focus of this paper is on the correct synchronization management in multithreaded programs. Similarly to previous work (see, e.g., [10]), we advocate the provision of a suitable package of software components that take care of the details of the thread synchronization, in a way that is transparent to the software developer. Among the programming languages supporting multithreading, we concentrate on Java. The reason is twofold. First, Java offers a set of mechanisms – like monitors and priorities – for the well-structured management of threads, which is portable on every system endowed with a Java Virtual Machine. Second, its object-orientation feature makes Java a natural candidate for the implementation of the above mentioned package, as a high level of abstraction from the details (encapsulation) can be achieved while supplying different levels of protection for objects and variables shared by several threads.

Our proposal is that both the implementation of the package and the use of its components for synchronizing the threads should be guided by the architecture of the software to be developed. In other words, what we propose is a software-architecture-driven management of the thread syn-

chronization within concurrent object-oriented programs. The goal of the software architecture level of design [9, 11] is that of producing a reference document, shared by all the people involved in the development of the software system, that describes the overall structure of the system as well as the functional and non-functional aspects of its overall behavior. Since the structure of the system is meant to be a collection of computational components together with a description of their interactions, we firmly believe that an architectural description of a concurrent object-oriented program is a well suited tool for achieving a correct management of the thread synchronization.

The thread synchronization model that we adopt encompasses two different dimensions. The first dimension is the thread communication mode and comprises three values: synchronous, asynchronous, asymmetric. By asymmetric communication we mean that one thread behaves synchronously with respect to the communication, whereas the other thread does not. The second dimension refers to the multiplicity of the communication and comprises three values: uni-uni, and-uni, or-uni. In a uni-uni communication, only two threads are involved (point-to-point). In an and-uni communication, a thread communicates with several other threads (broadcast). Finally, in an or-uni communication, a thread communicates with only one thread selected out of a set of other threads (client-server). Furthermore, the adopted thread synchronization model does not admit using global variables as well as passing object references while keeping a copy of the references, thus ruling out undesired side effects.

The development of the Java package – which we shall call `Sync` – for handling the synchronization of the Java threads and the data exchange among them adheres to the above mentioned synchronization model and is inspired by the main architectural abstractions. `Sync` is structured into four conceptual layers, each comprising a set of components realized through Java classes and interfaces having possibly different visibility degrees. The bottom-level layer, called `Connector`, is a set of Java classes and interfaces, which are not visible by the software developer, used within `Sync` to perform thread-to-thread synchronizations or data transfers. There are four classes realizing – consistently with the adopted synchronization model – the asynchronous-to-asynchronous, synchronous-to-asynchronous, asynchronous-to-synchronous, and synchronous-to-synchronous thread communications. The second layer, called `Port`, is a set of Java classes and interfaces, some which are visible, that realize the abstraction corresponding to a set of statements through which a thread interacts with possibly many other threads. There are twelve classes realizing – through instances of `Connector` – the uni-synchronous, and-synchronous, or-synchronous, uni-asynchronous, and-asynchronous, and or-asynchronous interactions both at the sending and at the receiving thread side. The third layer, called `ThreadElem`, is a single Java class derived from the `Thread` class, which realizes the abstraction corresponding to a thread in the concurrent Java program. The top-level layer, called `Architecture`, is a single Java class that provides support for coordinating the objects (of the three underlying layers) that will be instantiated when using `Sync`. [1]

As far as the use of `Sync` is concerned in the development of a concurrent Java program, according to our proposal this will be driven by the architectural level of design as well. More precisely, the starting point will be an architectural description of the concurrent Java program, which specifies at a high level of abstraction the program topology in terms of instances of threads and their communications based on the above mentioned synchronization model. In the literature, several architectural description languages (ADLs) have appeared, most of which are based on process algebra (see, e.g., [3, 8, 6, 5]). Among them, we choose the new version of PADL [1], as its synchronization model is very close to the adopted one. The idea is to automatically generate the multithreaded Java program from its architectural description in PADL, in such a way that all the synchronization details are transparently and correctly managed via `Sync`. This is carried out by means of a translator – which we shall call PADL2Java – that, given a PADL specification of a multithreaded Java program, creates as many instances of `ThreadElem`, `Port`, and `Connector` as needed to implement the multithreaded Java program. The use of PADL2Java and `Sync` will be exemplified by means of the design of a real-time audio processing system.

The paper is organized as follows. In Sect. 2 we illustrate the addressed problem and the motivation behind our approach by means of an example based on a real-time audio processing system, which is introduced together with its PADL description. In Sect. 3 we present the development of the Java package `Sync`. In Sect. 4 we show how PADL2Java translates PADL specifications into concurrent Java programs by means of `Sync`, which allows us to obtain an implementation of the real-time audio processing system. Finally, in Sect. 5 we report some concluding remarks.

2. Example: Real-Time Audio Processing

In this section we illustrate the addressed problem and the motivation behind our approach by introducing an example based on a simple real-time audio processing system.

1 Unlike [8] where a similar terminology is used within concurrent Java programs based on message passing, here the layers are not associated with specific communication modes.

The goal of this software system is to acquire and play a digital audio stream, allowing the user to change in real time the sound effects applied to the stream by means of a software sound processor. The audio processing system is organized as follows. The dry audio stream, coming from an input audio device, is modified by a sound processor according to some effect – like filtering or equalization – required by the user. Then the processed audio stream is forwarded to an output audio device, which plays it out. Both audio streams are sequences of audio samples. In order to avoid frequent accesses to the audio devices and to allow the sound processor to execute complex operations – like fast Fourier transform and convolution – that in general require several audio samples, the sequences of audio samples are segmented. The audio stream processing can be controlled by the user through a graphical user interface that interacts with the system console. More precisely, the user can start and stop the process and change the sound effect at audio processing time.

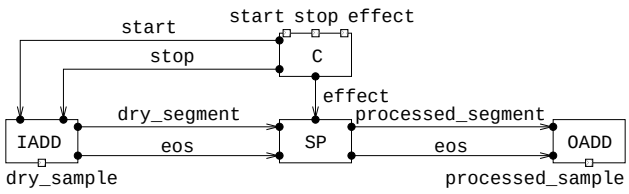

Figure 1. Audio processing system

As shown in Fig. 1, the considered software system is made out of four software components: the console, the input audio device driver, the sound processor, and the output audio device driver. Since the four software components work as independent entities that communicate to each other from time to time, it is natural to implement each of them as a thread. More precisely, the sound processor results in a CPU-bound thread, while the console and the two audio device drivers result in three I/O-bound threads. As far as the thread communication is concerned, only the console can interact with the user by receiving commands related to the start or the stop of the audio processing or the effect to be applied to the input audio stream. Likewise, only the two audio device drivers can interact with the two audio devices by exchanging audio samples. The console forwards the start and stop signals to the input audio device driver, while the effect is forwarded to the sound processor. The input audio device driver sends segments of dry audio samples to the sound processor and propagates to it the stop signals via end-of-stream signals. The sound processor in turn sends segments of processed audio samples to the output audio device driver and propagates to it the stop signals via end-of-stream signals.

In order to guarantee the quality of the played audio as well as the correct application of the effects, the four threads must be kept well synchronized. We assume that each of the two audio device drivers uses a standard audio port software utility, like e.g. the Java Media Framework, implementing an internal buffer. When the input audio device starts, the input buffer grows according to the sampling frequency of the input audio device. An input instruction is used by the input audio device driver to read from the input buffer an audio segment composed of a given number of dry samples. This input instruction blocks if it requires an audio segment that is longer than the number of unread dry audio samples contained in the input buffer. If no input is requested for a long time, the input buffer becomes full and the oldest unread dry audio samples are lost due to overwriting. Conversely, the output buffer decreases according to the sampling frequency of the output audio device, which is fed by the output audio device driver through an output instruction that appends to the output buffer an audio segment composed of a certain number of processed audio samples. This output instruction blocks if it tries to write an audio segment that is longer than the residual capacity of the output buffer. On the other hand, the output device driver can play out only if sufficiently many processed audio samples are present in the output buffer. Since the two instructions for interacting with the two audio devices are blocking, the two threads implementing the two audio device drivers must be kept synchronized at the same sampling frequency, and this synchronization has not to be disrupted by the sound processor, when applying an effect, or by the user via the console, when changing the effect. Keeping the two threads synchronized is possible if the effect processing time, which is the sum of the time to get the effect and the time to apply it to a segment, is less than the segment playout time (this is an obvious real-time constraint). Moreover, it is demanded that the output audio device driver introduces an artificial delay before playing out the first received audio segment, in order to compensate for a possible increase of the effect processing time incurred by subsequent audio segments to be played out.

Providing an architectural description of the considered software system, rather than directly implementing the multithreaded Java program, can be of paramount help to keep the thread synchronization under control. Based on this consideration, the architecture-driven approach that we propose aims at synthesizing well-synchronized multithreaded Java programs from their architectural specifications. In order to apply the approach to the real-time audio processing system illustrated above, we supply a high level description of the software system expressed in some ADL. Here we choose the new version [1] of PADL [5], as it conforms to the adopted synchronization model. Every PADL specification is composed of two sections. The first one contains the process algebraic definition of the behavior and the inter-

actions of the types of architectural elements – components and connectors – that are in the system. The second one contains the definition of the system topology, which is given as a set of declarations of architectural element type instances (depicted as boxes in Fig. 1), architectural interactions (depicted as white squares), and attachments (depicted as directed edges) among the non-architectural interactions (depicted as black circles) of the architectural element type instances.

The PADL description of the audio processing system starts with the identifier of the described architecture together with its constant parameters, which consist of the artificial delay of 125 msec introduced by the output audio device driver:

```
ARCHI_TYPE Audio_Processing_System(const integer
                                   delay := 125)
```

At the beginning of the first section of the PADL description – the one in which the behavior and the interactions of the types of architectural elements are specified – we have the definition of the console, which forwards to the appropriate components the user commands about the audio processing start/stop and the effect to be applied to the audio stream:

```
ARCHI_ELEM_TYPES
 ELEM_TYPE Console(void)
  BEHAVIOR
   Waiting_Start(void; void) =
    receive_start .
     forward_start . Waiting_Effect_Stop();
   Waiting_Effect_Stop(void; void) =
    choice
    {
     receive_effect .
      forward_effect . Waiting_Effect_Stop(),
     receive_stop .
      forward_stop . Waiting_Start()
    }
  INPUT_INTERACTIONS
   UNI SYNC receive_start; receive_effect;
            receive_stop
  OUTPUT_INTERACTIONS
   UNI SYNC forward_start; forward_effect;
            forward_stop
```

Note that all the interactions, which must be actions occurring in the behavior, are declared to be uni and synchronous.

Then we have the definition of the behavior and the interactions of the input audio device driver, which is responsible for opening/closing the input audio device according to the user commands and forwarding the segments of dry audio samples to the sound processor:

```
 ELEM_TYPE Input_Audio_Device_Driver(void)
  BEHAVIOR
   Idle(void; void) =
    receive_start . open_input_device . Busy();
   Busy(void; void) =
    choice
    {
     read_dry_samples .
      send_dry_segment . Busy(),
     receive_stop .
      close_input_device . send_eos . Idle()
    }
  INPUT_INTERACTIONS
   UNI SYNC receive_start; receive_stop
  OUTPUT_INTERACTIONS
   UNI SYNC send_dry_segment; send_eos;
            open_input_device; read_dry_samples;
            close_input_device
```

Afterwards we have the definition of the behavior and the interactions of the sound processor, which is in charge of applying the effect chosen by the user (or set by a suitable initialization) to the segments of dry audio samples in order to produce the audio stream to be played out:

```
ELEM_TYPE Sound_Processor(void)
 BEHAVIOR
  Init(void; void) =
   initialize . Receive_Effect_Segment();
  Receive_Effect_Segment(void; void) =
   choice
   {
    receive_dry_segment . process_dry_segment .
     send_processed_segment .
      Receive_Effect_Segment(),
    receive_effect . enable_effect .
     Receive_Segment(),
    receive_eos . forward_eos . Init()
   };
  Receive_Segment(void; void) =
   choice
   {
    receive_dry_segment . process_dry_segment .
     send_processed_segment .
      Receive_Effect_Segment(),
    receive_eos . forward_eos . Init()
   }
  INPUT_INTERACTIONS
   UNI SYNC receive_dry_segment; receive_effect;
            receive_eos
  OUTPUT_INTERACTIONS
   UNI SYNC send_processed_segment; forward_eos
```

Note that `initialize` and `process_dry_segment` are not interactions, but activities internal to the sound processor.

Finally we have the definition of the behavior and the interactions of the output audio device driver, which introduces a suitable delay when receiving the first segment of the processed audio stream and forwards such a stream to the audio device driver for being played out:

```
ELEM_TYPE Output_Audio_Device_Driver(const
                                     integer delay)
 BEHAVIOR
  Idle(void; void) =
   choice
   {
    receive_processed_segment .
     sleep(delay) . open_output_device .
      write_processed_samples . Busy(),
    receive_eos . Idle()
   };
  Busy(void; void) =
   choice
   {
```

```
      receive_processed_segment .
       write_processed_samples . Busy(),
      receive_eos . close_output_device . Idle()
     }
   INPUT_INTERACTIONS
    UNI SYNC receive_processed_segment;
             receive_eos
   OUTPUT_INTERACTIONS
    UNI SYNC open_output_device;
             write_processed_samples;
             close_output_device
```

We conclude with the second section of the PADL description, in which all the input interactions of the console and all the interactions of the two audio device drivers that deal with the two audio devices are declared to be the interfaces of the whole software system:

```
 ARCHI_TOPOLOGY
  ARCHI_ELEM_INSTANCES
   C    : Console();
   IADD : Input_Audio_Device_Driver();
   SP   : Sound_Processor();
   OADD : Output_Audio_Device_Driver(delay)
  ARCHI_INTERACTIONS
   C.receive_start; IADD.open_input_device;
   C.receive_effect; IADD.read_dry_samples;
   C.receive_stop; IADD.close_input_device;
   OADD.open_output_device;
   OADD.write_processed_samples;
   OADD.close_output_device
  ARCHI_ATTACHMENTS
   FROM C.start_sampling TO IADD.receive_start;
   FROM C.forward_effect TO SP.receive_effect;
   FROM C.stop_sampling TO IADD.receive_stop;
   FROM IADD.send_dry_segment
    TO SP.receive_dry_segment;
   FROM IADD.send_eos TO SP.receive_eos;
   FROM SP.send_processed_segment
    TO OADD.receive_processed_segment;
   FROM SP.forward_eos TO OADD.receive_eos
END
```

3. The Java Package `Sync`

In this section we describe the Java package `Sync`, which transparently handles the synchronization of the Java threads and the data exchange among them according to the adopted synchronization model. `Sync` is organized into four conceptual layers – `Connector`, `Port`, `ThreadElem`, `Architecture` – each of which corresponds to a different architectural abstraction and comprises a set of components realized through Java classes and interfaces.

The Java package `Sync` will be seen by the PADL2Java translator as a repository of architectural abstractions guaranteeing the correct and transparent handling of the synchronizations and data exchanges among the threads of the Java program being designed. In other words, starting from a PADL specification, the PADL2Java translator will be in charge of creating as many instances of `ThreadElem` as there are threads in the Java program, as many instances of `Port` as there are interactions in the `ThreadElem` instances, and all the instances of `Connector` that are needed to make the `ThreadElem` instances interact through their `Port` instances.

3.1. The Layer `Connector`

The layer `Connector` contains the bottom-level components of the package. `Connector` is a set of Java classes and interfaces, which are not visible by the software developer, that are used within `Sync` to perform thread-to-thread synchronizations or data transfers. The definition of the layer `Connector` is inspired by the producer-consumer model, where the producer and the consumer represent two different threads. This is realized by equipping every `Connector` class with a single buffer shared only by the related producer and consumer threads, together with two interface methods for accessing the buffer – `send()` and `receive()` – and four interface methods for observing the status of the threads using the `Connector` – `obsSndBlocking()`, `obsSndNotBlocking()`, `obsRcvBlocking()`, and `obsRcvNotBlocking()`.

The buffer can store a generic object that the producer transfers to the consumer. In the case of a data exchange the object represents some data structure, whereas in the case of a pure synchronization a null object is employed. In both cases, mutual exclusion must be enforced on the buffer accesses, i.e. at any time only one of the two involved threads can access the shared buffer. This is achieved by declaring the `send()` and `receive()` methods as synchronized methods, so that they result in a monitor-like control structure to which the primitive methods `wait()` and `notify()` can be applied.

The layer `Connector` comprises four classes realizing the following four thread-to-thread communications through the two synchronized interface methods `send()` and `receive()`: asynchronous-to-asynchronous, synchronous-to-asynchronous, asynchronous-to-synchronous, and synchronous-to-synchronous. The first communication type is the simplest one: neither `send()` nor `receive()` is blocking. The sending thread can immediately proceed after using `send()` to write a generic object into the buffer – if the previously written object has not been read in the meanwhile, then an exception is raised that propagates to the layer `Port`. Likewise, the receiving thread can proceed even if no new object has been read through `receive()` – an exception is raised whenever no new object has been read, which propagates to the layer `Port`. In the second communication type, `receive()` behaves similarly to the previous case, while `send()` invokes `wait()` – unless the receiving thread has previously tried to read from the buffer

without finding any new object – which causes the sending thread to block until the receiving thread wakes it up through an invocation to `notify()` contained in `receive()`. The third communication type is asymmetric as well, with the roles of the two participating threads being exchanged with respect to the synchronization: `receive()` invokes `wait()` if no new object has been found, while `send()` contains an invocation to `notify()` to wake up the receiving thread whenever necessary. Finally, in the fourth communication type, both `send()` and `receive()` contain invocations to both `wait()` and `notify()` in an order that complies with standard solutions to the synchronization problem for the producer-consumer system.

To conclude, the four interface methods `obsSndBlocking()`, `obsSndNotBlocking()`, `obsRcvBlocking()`, and `obsRcvNotBlocking()` are used by a receiving thread and a sending thread, respectively, to check for the willingness of other connected threads to communicate. More precisely, `obsSndBlocking()` and `obsRcvBlocking()` allow one thread to check the status of a connected thread and wait for it to be ready to communicate. Such two methods are therefore declared as synchronized methods. Instead, `obsSndNotBlocking()` and `obsRcvNotBlocking()` allow one thread to check the status of a connected thread (by returning a boolean value) without waiting for it to be ready to communicate. From the receiver viewpoint, an asynchronous sending thread is willing to communicate with it if it has already put into the buffer a new object to be read. From the sender viewpoint, an asynchronous receiving thread is willing to communicate with it if it has already tried to read a new object from the buffer without succeeding (which has caused an exception to be raised).

3.2. The Layer `Port`

The second layer, called `Port`, is a set of Java classes and interfaces that realize the abstraction corresponding to a set of statements through which a thread interacts with possibly many other threads. The layer `Port` comprises twelve classes realizing the uni-synchronous, and-synchronous, or-synchronous, uni-asynchronous, and-asynchronous, and or-asynchronous interactions both at the sending and at the receiving thread side. Every `Port` object instantiated by the PADL2Java translator will be attached to a single `Connector` object – if representing a uni-synchronous or a uni-asynchronous interaction – or several `Connector` objects – if representing another type of interaction.

A `Port` object must be instantiated by specifying in the `Port` constructor its owner thread, its priority, and its weight. In particular, information about the priority and the weight of the `Port` object can be set or got through the interface methods `setPriority()`, `getPriority()`, `setWeight()`, and `getWeight()`. In addition, another interface method is available – `ready()` – which is an utility that can be employed before proceeding with a communication to check whether the other parties are ready. This is achieved by `ready()` by invoking the method `obsSndNotBlocking()` or `obsRcvNotBlocking()` of each `Connector` object linking the `Port` object to another `Port` object.

Each of the six `Port` classes related to the sender side is equipped with a `send()` method, whereas each of the six `Port` classes related to the receiver side is equipped with a `receive()` method. Each of the two methods makes use of the homonymous method of the `Connector` classes, and raises an exception whenever it is invoked from within a `Port` object that is not attached to any other `Port` object via a `Connector` object. Another exception is propagated from the layer `Connector` to the layer `ThreadElem` whenever a `Port` object referring to an asynchronous receiving thread attempts to read from an empty buffer of a `Connector` object.

To conclude, we describe how the `send()` method works in each of the six cases (the `receive()` method works similarly):

- Uni-asynchronous: The sending `Port` object forwards the message via the non-blocking `send()` method of the only `Connector` object involving the `Port` object itself.
- And-asynchronous: The sending `Port` object polls the connected receiving `Port` objects through the `obsRcvNotBlocking()` method of the related `Connector` objects. If all the connected receiving `Port` objects are ready, then the sending `Port` object forwards the message via the non-blocking `send()` method of all the involved `Connector` objects. Otherwise, an exception is raised and no communication takes place.
- Or-asynchronous: The sending `Port` object polls the connected receiving `Port` objects through the `obsRcvNotBlocking()` method of the related `Connector` objects. If there is at least one connected receiving `Port` object that is ready, then the sending `Port` object forwards the message via the non-blocking `send()` method of the `Connector` object involving one of the ready connected receiving `Port` objects selected according to their priorities and weights. Otherwise, an exception is raised and no communication takes place.
- Uni-synchronous: The sending `Port` object forwards the message via the blocking `send()` method of the

only Connector object involving the Port object itself.

- And-synchronous: The sending Port object polls the connected receiving Port objects through the obsRcvBlocking() method of the related Connector objects. Then it forwards the message via the send() method of all the involved Connector objects.
- Or-synchronous: The sending Port object polls the connected receiving Port objects through the obsRcvNotBlocking() method of the related Connector objects. If there is at least one connected receiving Port object that is ready, then the sending Port object forwards the message via the send() method of the Connector object involving one of the ready connected receiving Port objects selected according to their priorities and weights. Otherwise, the sending Port object waits for one of the connected receiving Port objects to become ready.

3.3. The Layer ThreadElem

The third layer, called ThreadElem, is a single Java class derived from the Thread class. To achieve full compatibility, the constructors of the Thread class have been redefined in the ThreadElem class with the same parameters. Each ThreadElem object instantiated by the PADL2Java translator will thus correspond to a single thread in the concurrent Java program being designed.

The reason why a derived class has been defined instead of using the Thread class is that support must be provided to generate the Java code for a program thread from the behavior of the corresponding architectural element instance in the PADL specification. Since the generation of the pieces of code that are not related to the thread synchronization is outside the scope of this paper, we shall not provide a description of the supporting methods.

3.4. The Layer Architecture

The top-level layer, called Architecture, is a single Java class that provides support for coordinating the objects (of the three underlying layers) that will be instantiated when using Sync. In order to support hierarchical architectures, the Architecture class is derived from the ThreadElem class, hence it provides the same methods as the ThreadElem class and is fully compatible with it.

In order to hide the implementation details and avoid the difficulties deriving from the direct handling of the Connector objects, the Architecture class also defines an overloaded static method called attach(). The method attach() receives two parameters, which must be a sender Port object and a receiver Port object, and connects them only if they refer to two different owner threads and – according to the adopted synchronization model – result in a communication of the form uni-uni, and-uni, or-uni (should this not be the case, an exception would be raised). If the two Port object parameters are correct, the method attach() accomplishes its task in two steps. First, it creates a Connector object – and passes to it the references to the two Port objects – that realizes the connection between the two Port objects that conforms to their synchronicity and multiplicity features. Second, it passes a reference to the Connector object to both Port objects. Now the two Port objects can access their common Connector object, thus they can communicate through it using its methods.

4. The PADL2Java Translator

The main idea underlying our architecture-driven approach to the correct thread synchronization is to automatically generate the multithreaded program from its architectural description, in such a way that all the synchronization details are transparently and correctly managed via Sync. This is carried out by means of PADL2Java, a translator that, given a PADL specification of a concurrent software system, creates:

- as many instances of ThreadElem as there are architectural element type instances declared in the PADL specification;
- as many instances of Port as there are interactions in the architectural element type instances;
- all the instances of Connector that are needed to make the ThreadElem instances interact through their Port instances according to the attachments declared in the PADL specification.

The command to run the translator is the following:
PADL2Java [options] *file1*.padl *file2*.java
with the three following alternative options:

- Option -p, which is the default one, generates a .java file resulting in a Java program. This file contains a class derived from the Architecture one, a class derived from the ThreadElem one for each architectural element type in the .padl file, and a public class whose name is *file2* composed only of method main(). The latter class acts as a wrapper for the Architecture derived class by instantiating the derived Architecture class itself and starting the Architecture thread object, which in turn causes the classes of the underlying layer ThreadElem to be instantiated and the related thread objects to be suitably connected and then started. As far as the derived

ThreadElem classes are concerned, some comments are generated to remind the software developer to describe their behavior by means of a suitable sequence of Java instructions. Additional comments are generated within such classes for each of the architectural interactions of their instances, to remind the software developer to describe them through the handling of the related exceptions – the architectural interactions are not attached to other interactions, hence an exception is raised when trying to execute them as said in Sect. 3.2.

- Option -c generates a .java file resulting in a set of Java classes without any wrapper around them. More precisely, this file contains a public class derived from the Architecture one, whose name is *file2*, and a class derived from the ThreadElem one for each architectural element type in the .padl file. This option is useful to develop a software system in a hierarchical or a compositional way, by attaching the architectural interactions of a subarchitecture to the local (resp. architectural) interactions of other architectural element type instances (resp. subarchitectures).
- Option -a generates a .java file resulting in a Java applet. This file contains a class derived from the Architecture one, a class derived from the ThreadElem one for each architectural element type in the .padl file, and a public class derived from the Applet class whose name is *file2* composed only of the typical methods for activating, restoring, and deactivating applets. Like in the case of option -p, the latter class acts as a wrapper for the Architecture derived class. However, the instantiation of the derived Architecture class and the definition of the applet management methods are entrusted to the software developer, as reminded by suitable comments inserted into the methods themselves. This option is useful to embed the automatically generated code into HTML pages.

In any case, the automatically generated .java file imports the package Sync.

We now exemplify the use of PADL2Java and Sync by means of the design of the real-time audio processing system introduced in Sect. 2. The first class that we encounter in the generated file is the Architecture derived one, which is called Audio_Processing_System. Similarly to the declaration of the architectural topology in a PADL specification, this class contains several sections related to the architectural element type instances, their architectural interactions, and the attachments among them. In the first section of the class, a ThreadElem object is declared for each of the four architectural element type instances declared in the PADL specification of the audio processing system. Each such object belongs to a ThreadElem derived class – corresponding to an architectural element type in the PADL specification – which will be defined later in the generated file. In the second section, a public Port object is declared for each of the nine architectural interactions declared in the PADL specification of the audio processing system. Such objects are declared to be public because from a conceptual viewpoint the architectural interactions are the interfaces of the whole architecture, hence support must be provided for them to be used for the hierarchical or compositional modeling of complex architectures. In the third section, the class constructor is defined together with its parameters, which coincide with the parameters of the architecture element type. The constructor just invokes the method defined in the next section. In the fourth section, a method called buildArchiTopology() is defined, which builds up the architecture topology. This methods instantiates the four previously declared ThreadElem objects, assigns the nine previously declared public Port objects through the corresponding Port objects of the newly instantiated ThreadElem objects, and invokes method attach() seven times to connect the Port objects of the newly instantiated ThreadElem objects according to the seven attachments declared in the PADL specification of the audio processing system. Finally, in the fifth section, a method called run() is defined to make the homonymous Thread method concrete. This method starts the execution of the four instantiated ThreadElem objects, then waits for their termination.

```
class Audio_Processing_System
                      extends Architecture {
 //------------- DECLARING THREADS ------------//
 Console C;
 Input_Audio_Device_Driver IADD;
 Sound_Processor SP;
 Output_Audio_Device_Driver OADD;
 //--- DECLARING ARCHITECTURAL INTERACTIONS ---//
 public UniSyncReceiverPort receive_start;
   ...
 public UniSyncSenderPort close_output_device;
 //----------- DEFINING CONSTRUCTOR -----------//
 protected int delay;
 Audio_Processing_System(int delay) {
  this.delay = delay;
  buildArchiTopology();
 }
 //----------- BUILDING ARCHITECTURE ----------//
 void buildArchiTopology() {
  //----- ARCHITECTURAL ELEMENT INSTANCES ----//
  C = new Console();
  IADD = new Input_Audio_Device_Driver();
  SP = new Sound_Processor();
  OADD = new Output_Audio_Device_Driver(delay);
  //------- ARCHITECTURAL INTERACTIONS -------//
  this.receive_start = C.receive_start;
    ...
  this.close_output_device =
   OADD.close_output_device;
  //-------- ARCHITECTURAL ATTACHMENTS -------//
  try {
   attach(C.start_sampling, IADD.receive_start);
```

```
    ...
   attach(SP.forward_eos, OADD.receive_eos);
  } catch (BadAttachmentException e) {}
 }
 //----------- RUNNING ARCHITECTURE  ----------//
 public void run() {
  C.start(); IADD.start();
  SP.start(); OADD.start();
  try {
   C.join(); IADD.join(); SP.join(); OADD.join();
  } catch (InterruptedException e) {}
 }
}
```

Afterwards, in the generated file we encounter a `ThreadElem` derived class for each of the four architectural element types defined in the PADL specification of the audio processing system. For the sake of brevity, we show only the class generated for the output audio device driver type. Similarly to the definition of an architectural element type in a PADL specification, this class contains several sections related to the behavior definition and the declaration of its input and output interactions. In the behavior section of the class, some comments are generated to remind the software developer to define both the methods that characterize the behavior itself – which will be invoked within the method `run()` – and the exception handling for each possible architectural interaction of the instances of the architectural element type associated with the class. As far as the interactions are concerned, we point out that they are implemented through `Port` objects that are public, as they must be visible in the `Architecture` derived class of the generated file. We further observe that a non-default constructor for the class is defined only if the associated architectural element type has at least one parameter.

```
class Output_Audio_Device_Driver
                         extends ThreadElem {
 //----------- DEFINING CONSTRUCTOR -----------//
 protected int delay;
 Output_Audio_Device_Driver(int delay) {
  this.delay = delay;
 }
 //------------- DEFINING BEHAVIOR ------------//
 public void run() {
  // TODO - INVOCATION OF BEHAVIOR METHODS
 }
  // TODO - DEFINITION OF BEHAVIOR METHODS
  // TODO - DEFINITION OF EXCEPTION HANDLERS FOR:
     // open_output_device.send()
     // write_processed_samples.send()
     // close_output_device.send()
 //----- INSTANTIATING INPUT INTERACTIONS -----//
 public UniSyncReceiverPort
  receive_processed_segment =
  new UniSyncReceiverPort(this);
 public UniSyncReceiverPort receive_eos =
  new UniSyncReceiverPort(this);
 //----- INSTANTIATING OUTPUT INTERACTIONS ----//
 public UniSyncSenderPort open_output_device =
  new UniSyncSenderPort(this);
 public UniSyncSenderPort
  write_processed_samples =
  new UniSyncSenderPort(this);
 public UniSyncSenderPort close_output_device =
  new UniSyncSenderPort(this);
}
```

If option -p is used, i.e. a full Java program must be synthesized, a further class is appended to the generated file. This class, which is public and named *file2*, contains only the method `main()`. This method acts as a wrapper for the `Architecture` derived class `Audio_Processing_System` by instantiating and starting it, then waiting for it to terminate.

```
public class <file2> {
 public static void main(String args[]) {
  //------- INSTANTIATING ARCHITECTURE -------//
  int delay = 125;
  Audio_Processing_System archiInstance =
   new Audio_Processing_System(delay);
  //---------- RUNNING ARCHITECTURE ----------//
  archiInstance.start();
  try { archiInstance.join(); }
  catch (InterruptedException e) {}
 }
}
```

If option -c is used instead, then no wrapper class has to be appended to the generated file. However, the `Architecture` derived class `Audio_Processing_System` is declared to be public, in order for it to be visible to support the design of complex software architectures specified in a hierarchical or compositional way.

Finally, if option -a is used, i.e. a Java applet must be synthesized, a further class is appended to the generated file. This class, which is public and named *file2*, is derived from the `Applet` class and contains several sections that result in a wrapper for the `Architecture` derived class `Audio_Processing_System`. In the first section of the class, an object of class `Audio_Processing_System` is declared. Note that, unlike the program wrapper, here the previously mentioned object is not instantiated, as this will be done by the software developer within a suitable method defined in the third section. In the second section, a `Port` object is declared for each of the nine architectural interactions declared in class `Audio_Processing_System`. These can be attached to the architectural interactions and then used by the software developer within suitable methods defined in the third section. The idea behind such additional interactions is to allow the applet wrapper to dispatch suitable commands to the right threads in the architecture depending on the events that are caught. Each of the additional interactions is asynchronous – hence it cannot block the applet wrapper – and uni – hence it can surely be attached to the corresponding architectural interaction if necessary. In order to help the software developer to correctly attach the additional interactions to the corresponding architectural interactions, each additional interaction has the same name as the corresponding architectural interaction

preceded by the prefix `to_` (resp. `from_`) if the corresponding architectural interaction is an input (resp. output) interaction. Finally, in the third section the stubs for the typical methods of the `Applet` class are added, with some comments in them to remind the software developer to define them if their behavior is different from the default one. Such methods are related to the activation, restoration, and deactivation of the applets.

```
public class <file2> extends Applet {
 //---------- DECLARING ARCHITECTURE ----------//
 int delay = 125;
 Audio_Processing_System archiInstance;
 //---------- DECLARING APPLET PORTS ----------//
 UniAsyncSenderPort to_receive_start;
   ...
 UniAsyncReceiverPort from_close_output_device;
 //---------- DEFINING APPLET MEMBERS ---------//
 public void init() {
  // TODO - POSSIBLE METHOD DEFINITION
 }
 public void start() {
  // TODO - POSSIBLE METHOD DEFINITION
 }
 public void paint() {
  // TODO - POSSIBLE METHOD DEFINITION
 }
 public void stop() {
  // TODO - POSSIBLE METHOD DEFINITION
 }
 public void destroy() {
  // TODO - POSSIBLE METHOD DEFINITION
 }
}
```

5. Conclusion

In this paper we have proposed a software-architecture-driven approach to the correct thread synchronization within concurrent Java programs. The approach relies on the definition of a Java package inspired by the main architectural abstractions, which is transparently used by a translator that automatically synthesizes multithreaded Java programs from their architectural specifications in PADL. For a complete evaluation of the approach, further case studies – besides the audio processing system – have to be carried out, so that indicators like the average percentage of code that can be automatically generated and its impact on the system efficiency can be assessed. For the future we plan to extend the approach in order to take into account the automatic generation of the thread behavior as well, to investigate to what extent the properties proved at the architectural level by means of the analysis techniques for PADL described in [5, 1] are preserved by the automatically generated code, and to integrate the approach in the software tool TwoTowers [4].

As far as related work is concerned, first of all we mention ArchJava [2]. This 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 specification with respect to communication integrity. Our approach differs from ArchJava as it mainly focuses on the correct thread synchronization issue with respect to a rich synchronization model implemented and transparently made available through `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). The second difference is that our approach keeps the ADL and the implementation language separated, in order to be able to prove more complex properties that could not be demonstrated on a Java-like language. We also would like to mention [7], where an approach is proposed to automatically generate Java code from SDL specifications. Although the target of this approach and ours is similar, the framework in which the two approaches are considered is quite different. In fact, while our approach deals with the correct thread synchronization within a single Java program, the other approach aims at the generation of distributed Java code whose components are coordinated via CORBA.

References

[1] A. Aldini and M. Bernardo, *"A General Deadlock Detection Approach for Software Architectures"*, in Proc. FME 2003, LNCS 2805:658-677.

[2] J. Aldrich, C. Chambers, and D. Notkin, *"ArchJava: Connecting Software Architecture to Implementation"*, in Proc. of ICSE 2002, IEEE-CS Press.

[3] R. Allen and D. Garlan, *"A Formal Basis for Architectural Connection"*, in ACM TOSEM 6:213-249, 1997.

[4] M. Bernardo, *"TwoTowers 3.0: Enhancing Usability"*, in Proc. of MASCOTS 2003, IEEE-CS Press, pp. 188-193.

[5] M. Bernardo, P. Ciancarini, and L. Donatiello, *"Architecting Families of Software Systems with Process Algebras"*, in ACM TOSEM 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] R. Guimarães and W. Borelli, *"An Automatic Java Code Generation Tool for Telecom Distributed Systems"*, in Proc. of SOFTCOM 2002.

[8] J. Magee and J. Kramer, *"Concurrency: State Models & Java Programs"*, Wiley, 1999.

[9] D.E. Perry and A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT SEN 17:40-52, 1992.

[10] A. Poggi and G. Rimassa, *"An Efficient and Flexible C++ Library for Concurrent Programming"*, in Software Practice and Experience 28:1437-1463, 1998.

[11] M. Shaw and D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996.