PADL2Java: A Java Code Generator for Process Algebraic Architectural Descriptions

Edoardo Bontà and Marco Bernardo
Università di Urbino “Carlo Bo” – Italy
Istituto di Scienze e Tecnologie dell’Informazione

Abstract

One of the main objectives of model-driven software engineering is to produce code automatically from high-level design models. This goal can be achieved by providing suitable models and model-to-code transformations that ensure the conformance of the produced code to its high-level specification. In this context we have developed PADL2Java, a software tool that translates PADL models into Java code. PADL is a process algebraic architectural description language equipped with a rigorous semantics and transformation rules into multithreaded object-oriented software, which is employed in the verification tool TwoTowers. This paper discusses the code generation approach underlying PADL2Java, the structure of the synthesized code, and the integration of the translator in TwoTowers. The effectiveness of PADL2Java is illustrated through the generation of a Java implementation of a cruise control system.

1. Introduction

The adoption of high-level design models for producing software artifacts such as code, documentation, and other deliverables has become a common practice for software engineers in the last few years. Nevertheless, the model-driven approach requires attention in selecting the appropriate notation – by taking care of its formal rigor and usability – when building models for generating code. Adequate models and model-to-code transformations should be conceived and the produced code should conform to its high-level specification.

Several architectural description languages have been proposed for the purpose of modeling software. Many of them, like Wright [4, 3], Darwin/FSP [21, 22], LEDA [11], PADLÆmilia [1], and π-ADL [25], are based on process algebra [24, 18, 5, 6]. The reason is that process algebra provides a formal support to compositional modeling and its analysis techniques can be adapted to the verification of architectural mismatch freedom [4, 19, 12, 1]. From a process algebraic perspective, these languages are an improvement in terms of usability, as they highlight architectural concepts like components, connectors, and styles while hiding process algebra technicalities to the designer.

Process algebraic architectural description languages support automated code generation by means of techniques that translate architectural specifications into multithreaded object-oriented programs. Based on previous work [22, 8, 9, 10], we have developed PADL2Java, a software tool that translates PADL descriptions into Java code. This paper discusses the code generation approach underlying PADL2Java, the structure of the synthesized code, and the integration of the translator in TwoTowers. The effectiveness of PADL2Java is illustrated through the generation of a Java implementation of a cruise control system examined in [20, 1].

This paper is organized as follows. In Sects. 2 and 3 we motivate the choice of PADL as source language and of Java as target language, respectively. In Sect. 4 we exhibit the transformation approach on which PADL2Java is based. In Sect. 5 we illustrate the structure of the generated code by showing the result of the application of PADL2Java to the PADL description of the cruise control system mentioned above. In Sect. 6 we present the integration of PADL2Java in TwoTowers. Finally, in Sect. 7 we provide some concluding remarks and directions for future work.

2. The Source Specification Language

Our source specification language is PADL [1]. The reason for concentrating on it among the various process algebraic architectural description languages appeared in the literature is that PADL achieves a reasonable balance among usability, expressiveness, and analyzability. Moreover, it is equipped with TwoTowers [7], a software tool for the functional verification, performance evaluation, and security analysis of process algebraic architectural descriptions expressed in its performance-aware variant Æmilia.
A PADL description represents an architectural type, which is a family of software systems sharing certain constraints on the observable behavior of their components as well as on their topology. As shown in Table 1, the textual description of an architectural type in PADL starts with its name and its formal parameters (initialized with default values), then comprises an architectural behavior section and an architectural topology section.

The first section defines the overall behavior of the system family by means of types of software components and connectors, which are collectively called architectural element types. The definition of an AET, which starts with its name and its formal parameters, consists of the specification of its behavior and of its interactions.

The behavior of an AET has to be provided in the form of a sequence of behavioral equations written in a verbose variant of process algebra allowing only for the inactive process (rendered as \texttt{stop}), the action prefix operator supporting possible boolean guards and value passing, the alternative composition operator (rendered as \texttt{choice}), and recursion.

Interactions are actions occurring in the process algebraic specification of the behavior of the AET that act as interfaces for the AET itself, while all the other actions are assumed to represent internal activities. Each interaction has to be equipped with three qualifiers, with the first qualifier establishing whether the interaction is an input or output interaction.

The second qualifier represents the synchronicity of the communications in which the interaction can be involved. We distinguish among synchronous interactions which are blocking (default qualifier \texttt{SYNC}), semi-synchronous interactions which are completely decoupled from the other parties involved in the communication (qualifier \texttt{ASYNC}).

The third qualifier describes the multiplicity of the communications in which the interaction can be involved. We distinguish among uni-interactions which are mainly involved in one-to-one communications (qualifier \texttt{UNI}), and/or-interactions guiding inclusive one-to-many communications (qualifier \texttt{AND}), and or-interactions guiding selective one-to-many communications (qualifier \texttt{OR}). It can also be established that an output or-interaction depends on an input or-interaction, in order to guarantee that a selective one-to-many output is sent to the same element from which a selective many-to-one input was received (keyword \texttt{DEP}).

The second section of a PADL description defines the topology of the system family. This is accomplished in three steps. Firstly, we have the declaration of the instances of the AETs – called AEIs – which represent the actual system components and connectors, together with their actual parameters. Secondly, we have the declaration of the architectural (as opposed to local) interactions, which are some of the interactions of the AEIs that act as interfaces for the whole systems of the family. Thirdly, we have the declaration of the architectural attachments among the local interactions of the AEIs, which make the AEIs communicate with each other. An attachment is admissible only if it goes from an output interaction of an AEI to an input interaction of another AEI. Moreover, a uni-interaction can be attached only to one interaction, whereas an and/or-interaction can be attached only to uni-interactions.

### Table 1. Structure of a PADL textual description

<table>
<thead>
<tr>
<th>ARCHI_TYPE</th>
<th>&lt;name and initialized formal parameters&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCHI_BEHAVIOR</td>
<td>: :</td>
</tr>
<tr>
<td>ARCHI_ELEM_TYPE</td>
<td>&lt;AET name and formal parameters&gt;</td>
</tr>
<tr>
<td>BEHAVIOR</td>
<td>&lt;sequence of process algebraic equations built from stop, action prefix, choice, and recursion&gt;</td>
</tr>
<tr>
<td>INPUT_INTERACTIONS</td>
<td>&lt;input synchronous/semi-synchronous/asynchronous uni/and/or-interactions&gt;</td>
</tr>
<tr>
<td>OUTPUT_INTERACTIONS</td>
<td>&lt;output synchronous/semi-synchronous/asynchronous uni/and/or-interactions&gt;</td>
</tr>
<tr>
<td>: :</td>
<td></td>
</tr>
<tr>
<td>ARCHI_TOPOLOGY</td>
<td></td>
</tr>
<tr>
<td>ARCHI_ELEM_INSTANCES</td>
<td>&lt;AEI names and actual parameters&gt;</td>
</tr>
<tr>
<td>ARCHI_INTERACTIONS</td>
<td>&lt;architecture-level AEI interactions&gt;</td>
</tr>
<tr>
<td>ARCHI_ATTACHMENTS</td>
<td>&lt;attachments between AEI local interactions&gt;</td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>
The semantics for PADL is given by translation into process algebra. Basically, the semantics of every AEI is the sequence of process algebraic equations defining the behavior of the corresponding AET. Then, the semantics of an entire architectural description is the parallel composition of the semantics of the constituent AEIs, with synchronization sets determined by the attachments.

2.1. Describing a Cruise Control System

We now illustrate PADL by formalizing the cruise control system considered in [20, 1]. This software system has to be embedded into a car with standard accelerator and brake pedals. On the hardware side, we assume that the interaction between the driver and the cruise control system is realized by means of three buttons: on, off, and resume. When on is pressed, the cruise control system records the current speed and then maintains the automobile at that speed. When the accelerator, the brake, or off is pressed, the cruise control system disengages but retains the speed setting. If resume is pressed later on, then the system is able to accelerate or decelerate the automobile back to the previously recorded speed. The cruise control system has to be designed in such a way that deadlock cannot occur.

First of all, we observe that we need at least four types of software components: a sensor, a speed controller, a speed detector, and a speed actuator. All of them will be defined in the section ARCHI_BEHAVIOR of the PADL description of an architectural type that we call **Cruise Control**.

The sensor detects the driver’s commands – turning the engine on/off, pressing the accelerator/brake, and pressing one of the on/off/resume buttons – and forwards them to the speed controller. The sensor AET is defined as follows:

```plaintext
ARCHI_ELEM_TYPE Sensor_Type(void)
BEHAVIOR
Sensor_Off(void; void) = turn_engine_on . Sensor_On();
Sensor_On(void; void) =
  choice
  |
  pressed_accelerator . Sensor_On(),
  pressed_brake . Sensor_On(),
  pressed_on . Sensor_On(),
  pressed_off . Sensor_On(),
  pressed_resume . Sensor_On(),
  turn_engine_off . Sensor_Off()
}
INPUT_INTERACTIONS void
OUTPUT_INTERACTIONS SYNC UNI
SYNC AND turn_engine_on;
turn_engine_off
```

The speed controller triggers the speed actuator on the basis of the driver’s commands received through the sensor. It can be in one of the following four states: inactive (when the engine is off), active (when the engine is on), cruising (after pressing the on button in the active state or the resume button in the suspended state), and suspended (after pressing any pedal or button different from on/resume in the cruising state). The controller AET is defined as follows:

```plaintext
ARCHI_ELEM_TYPE Controller_Type(void)
BEHAVIOR
Inactive(void; void) =
Active(void; void) =
  choice
  |
  pressed_accelerator . Active(),
  pressed_brake . Active(),
  pressed_on . trigger_record . Cruising(),
  pressed_off . Active(),
  pressed_resume . Active(),
  turn_engine_off . Inactive();
Cruising(void; void) =
  choice
  |
  pressed_accelerator . trigger_disable . Suspended(),
  pressed_brake . trigger_disable . Suspended(),
  pressed_on . trigger_record . Cruising(),
  pressed_off . Suspended(),
  pressed_resume . trigger_enable . Cruising(),
  turn_engine_off . trigger_disable . Inactive();
Suspended(void; void) =
  choice
  |
  pressed_accelerator . Suspended(),
  pressed_brake . Suspended(),
  pressed_on . Suspended(),
  press_off . Suspended(),
  pressed_resume . trigger_enable . Cruising(),
  turn_engine_off . trigger_disable . Inactive();
}
INPUT_INTERACTIONS SYNC UNI
turn_engine_on;
turn_engine_off;
pressed_brake;
powered_on;
powered_off;
powered_resume
OUTPUT_INTERACTIONS SYNC UNI trigger_enable;
trigger_disable;
trigger_record
```

The speed detector periodically communicates the number of wheel revolutions per time unit to the speed actuator. The speed detector AET is defined as follows:

```plaintext
ARCHI_ELEM_TYPE Detector_Type(void)
BEHAVIOR
Detector_Off(void; void) =
Detector_On(void; void) =
  choice
  |
  measure_speed . signal_speed . Detector_Off(),
  turn_engine_off . Detector_Off();
}
INPUT_INTERACTIONS SYNC UNI
turn_engine_on;
turn_engine_off;
powered_brake;
powered_on;
powered_off;
powered_resume
OUTPUT_INTERACTIONS SYNC UNI signal_speed
```

The speed actuator adjusts the throttle on the basis of the triggers received from the controller and of the speed measured by the detector. It can be in one of the following two states: disabled (until the on/resume button is pressed) and enabled (until any pedal or button different from on/resume is pressed). The speed actuator AET is defined as follows:
ARCHI_ELEM_TYPE Actuator_Type(void)

BEHAVIOR

Disabled(void; void) =
choice
| signalled_speed . Disabled(),
triggered_enable . enable_speed_ctrl . Enabled(),
triggered_record . record_speed . Enabled() |
| Enabled(void; void) =
choice
| signalled_speed . adjust_throttle . Enabled(),
triggered_disable . disable_speed_ctrl . Disabled() |

INPUT_INTERACTIONS SYNC UNI signalled_speed;
triggered_enable;
triggered_disable;
triggered_record

OUTPUT_INTERACTIONS void

Finally, the section ARCHI_TOPOLOGY contains the declaration of the four software components together with the attachments among their local interactions, which result in a cyclic topology:

ARCHI_ELEM_INSTANCES
S : Sensor_Type();
C : Controller_Type();
D : Detector_Type();
A : Actuator_Type()

ARCHI_INTERACTIONS
void

ARCHI_ATTACHMENTS
FROM S.turn_engine_on TO C.turned_engine_on;
FROM S.turn_engine_on TO D.turned_engine_on;
FROM S.turn_engine_off TO C.turned_engine_off;
FROM S.turn_engine_off TO D.turned_engine_off;
FROM S.press_accelerator TO C.pressed_accelerator;
FROM S.press_brake TO C.pressed_brake;
FROM S.press_on TO C.pressed_on;
FROM S.press_off TO C.pressed_off;
FROM S.press_resume TO C.pressed_resume;
FROM C.trigger_enable TO A.triggered_enable;
FROM C.trigger_record TO A.triggered_record;
FROM D.signal_speed TO A.signalled_speed

In [1] it has been verified through the architectural interoperability check that the PADL description shown above is deadlock free, as initially requested.

3. The Target Programming Language

Our target programming language is Java. As in [22], this choice has been made for the following two reasons. First, Java supports multithreading and offers a set of mechanisms for the well-structured management of threads and their shared data, which should simplify the code generation task given the concurrency inherent in process algebraic architectural descriptions.

Second, its object-oriented nature – and specifically its encapsulation capability – makes Java an appropriate candidate for coping with the high level of abstraction typical of process algebraic architectural description languages. In fact, as can be expected, the translation of PADL descriptions into Java software cannot be complete, and hence will require the intervention of the software developer in specific positions of the generated code, e.g. for inserting the Java statements corresponding to internal actions.

Another good reason for selecting Java is that it is equipped with software model-checking tools, like e.g. Java PathFinder [26]. These tools complement the analysis conducted on PADL descriptions by making it possible the verification of property preservation at the code level. In fact, although property preservation is guaranteed under certain constraints [8, 9], an inappropriate intervention of the software developer on the generated code may lead to the violation of properties proved at the architectural level.

4. The Code Generation Approach

Among the various approaches for automatically generating code from architectural descriptions, we can distinguish two families on the basis of the distance between the formalism used for describing software architectures (in our case, PADL) and the implementation language in which code is generated (in our case, Java).

The first family, characterized by an exogenous transformation, is the long-distance one. In this family, the formalism is kept well separated from the implementation language, and descriptions are entirely translated into code. To this family belong architectural description languages endowed with code generation facilities such as Aesop [16], C2SADEL [23], and Darwin [21].

The second family, characterized by a semi-endogenous transformation, is the short-distance one. In this family, the architectural formalism is embedded in the implementation code in the form of special comments, as in SyncGen [14], or in the form of special keywords and statements, as in ArchJava [2]. Here, only special symbols are translated into implementation code, whereas the rest is left unchanged.

While the latter transformation can offer a flexible support to software developers – as classical programming techniques and patterns can be applied when designing systems – the level of abstraction of the underlying architectural formalism is usually low. In the case of process algebraic architectural description languages, the transformation typically adopted for generating code is the former. The reason is that such languages are specifically conceived for abstracting high-level properties of entire software systems, hence they are distant from implementation languages.

One of the ideas at the basis of the approach of PADL2Java is the provision of a library of software components – a package called Sync – for adding architectural capabilities to the target programming language in order to shorten its distance from the architectural description language from which the code will be generated. Hence, the approach underlying our translator can be viewed as
Figure 1. Class diagrams of the conceptual layers of Sync
a semi-exogenous transformation, as opposed to the semi-
endogenous approach.

On the basis of the classification given in [13], we
can say that if the target model is considered as a text, a
semi-endogenous transformation can be easily realized as
a model-to-text, template-based transformation. Unfortu-
nately, this transformation cannot be applied with the same
facility to our semi-exogenous approach, as the distance
from model to text is still long. However, thanks to pack-
age Sync, a model-to-text, visitor-based transformation has
been implemented very easily in PADL2Java.

We now discuss the two transformation models on which
PADL2Java relies together with the implementation of the
translator itself and in particular of package Sync.

4.1. Transformation Models

Two models have been considered for treating separately
the thread communication management and the thread be-
behavior management.

Thread Communication Model. The first model [8] is
strongly based on the availability of the package Sync on
the target side, which provides a set of Java classes and in-
faces for handling communications and for implement-
ing AETs as threads. As shown in Fig. 1, package Sync
is structured into four conceptual layers, each correspond-
ing to a different architectural abstraction. In particular,
the Sync-based code generated for thread communication
complies with the communication model defined for PADL,
which relies on two roles (sender and receiver) and encom-
passes two different dimensions.

The first dimension is the communication synchronic-
ity and comprises nine values: synchronous to syn-
chronous, synchronous to semi-synchronous, synchronous
to asynchronous, semi-synchronous to synchronous, semi-
synchronous to semi-synchronous, semi-synchronous to
asynchronous, asynchronous to synchronous, asynchronous
to semi-synchronous, and asynchronous to asynchronous.
In the synchronous case, the thread waits for the other to
become ready. In the semi-synchronous case, the thread
checks whether the other is ready and, if not, raises an ex-
ception without blocking. In the asynchronous case, the
thread simply sends/receives a signal or message through a
buffer and then proceeds independently of the status of the
other thread (an exception is raised at the asynchronous re-
ceiving side if no signal/message is available in the buffer).

The second dimension refers to the communication mul-
tiplicity and comprises three values: uni-uni, and-uni, or-
uni. In a uni-uni communication, only two threads are in-
volved (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
(server-clients).

Thread Behavior Model. In the second model [9], the only
process algebraic operators admitted in the behavioral equa-
tions of an AET – inactive process (stop), action prefix, al-
ternative composition (choice), and recursion – are trans-
lated into Java code. As already mentioned, the translation
into a thread cannot be completely automated, due to the
different level of abstraction of an architectural description
language with respect to a programming language.

In particular, a different treatment is needed for the ac-
tion prefix operator depending on whether the action –
which corresponds to a sequence of thread statements – is
an internal action or an interaction. While the latter is in-
volved in communications, hence it is managed as the com-
munication model prescribes, the former can only be ren-
dered as a stub during the translation of the thread behavior,
which will have to be manually filled in by the developer.

4.2. Implementation of PADL2Java

The translator PADL2Java is composed of a set of Java
classes created by the parser generator JavaCC from the
grammar specification of PADL. Other classes, based on
the visitor pattern [15], have been developed for inspect-
ing and transforming the internal representation of a parsed
PADL description and for generating code from it. In fact,
as already mentioned, PADL2Java follows a model-to-text,
visitor-based approach for generating code.

Figure 2. PADL2Java visitor class hierarchy

In order to facilitate the development of the visitor
classes, a further template-based transformation has been
adopted during the implementation of PADL2Java. Since
JavaCC generates a visitor interface in which an overloaded
method visit() is declared for each different node type
of the internal representation, a very simple tool called vis-
itorExpander has been developed to be used in conjunction
with JavaCC. This tool is in charge of generating concrete
visitor classes from the visitor interface and from a template
where a single generic method \texttt{visit()} is defined – together with possible additional utility methods. The generic
method is expanded for each of the overloaded methods declared in the interface. Each of the generated classes can
then be used as a base for more advanced visitor classes in which only a subset of the \texttt{visit()} methods needs to be
overridden, depending on the specific task to be performed.

In particular, two concrete base classes have been
generated with the tool visitorExpander, as illustrated
in the intermediate level of Fig. 2. The first one,
ExplorerVisitorBase, which simply explores the in-
ternal representation, has been employed as a base class for
developing a family of inspector visitors and code generator
visitors. The second one, DuplicatorVisitorBase,
which duplicates each node of the internal representation
encountered during its visit, has been used instead as a base
class for developing a family of manipulator visitors. The
rules adopted by the two families of visitors are those pre-
scribed by the two models of Sect. 4.1.

5. Structure of the Generated Code

The structure of the code generated by PADL2Java is il-
lustrated in Fig. 3. As shown in the upper part of the
figure, PADL2Java synthesizes a class implementing the
interface RunnableArchi plus as many classes imple-
menting the interface RunnableElem as there are AETs
in the PADL description. We point out that, in order to
provide a Java abstraction for representing hierarchical or
composite architectural types, within Sync the interface
RunnableArchi has been defined by extending the inter-
face RunnableElem, which extends in turn the standard
interface Runnable.

In the lower part of Fig. 3 shows the stub classes to be
manually filled in. These are generated for man-
aging internal actions of AETs implemented as thread
classes (IAS) and for handling exceptions that may
be raised by interactions implemented within thread
classes (EHS), e.g. UnattachedPortException and
NotReadyPortException. Fig. 3 also shows a pack-
age called Structs, which handles those PADL data types
that cannot be translated into built-in Java data types.

Each of the classes generated by PADL2Java for a given
PADL description is stored into a distinct .java file. All
these files are contained in a single package with the same
name as the PADL description source file. Further .java
files may be generated depending on the translation option
specified upon invoking PADL2Java (default option -c).

If option \texttt{-p} is used, a full Java program is synthesized.
This is done by PADL2Java through the generation of a fur-
ther public class containing only method main(), which
acts as a wrapper for the RunnableArchi-implementing
class. Instead, if option \texttt{-a} is used, a Java applet is syn-
thesized. This is done by PADL2Java through the gen-
eration of a further public class derived from the stan-
dard JApplet class, which results in a wrapper for the
RunnableArchi-implementing class.

5.1. Generating the Cruise Control System

We now illustrate the use of PADL2Java by means of the
synthesis of Java code for the cruise control system de-
scribed with PADL in Sect. 2.1. Due to lack of space, we
only show the translation of AET Detector,Type and of
section ARCHI_TOPOLOGY of the PADL description.

The former is generated as a thread class in which the
core section DEFINING BEHAVIOR translates each orig-
inal PADL behavioral equation into a homonymous in-
stance of an anonymous class implementing the interface
BehavioralEquationInterface:

```java
class Detector_Type implements RunnableElem {
    // DECLARING BEHAVIORAL EQUATIONS INTERFACES
    interface BehavioralEquationInterface {
        void behavEqCall();
    }
    BehavioralEquationInterface Detector_Off, Detector_On;
    BehavioralEquationInterface nextBehavEq;
}
```

![Figure 3. Generated Java files](image-url)
Object[] actualPars;
//-------- INSTANTIATING INTERACTIONS --------//
SyncUniReceiverPort turned_engine_on =
    new SyncUniReceiverPort(this);
SyncUniReceiverPort turned_engine_off =
    new SyncUniReceiverPort(this);
SyncUniSenderPort signal_speed =
    new SyncUniSenderPort(this);
//--------------- DECLARING STUBS --------------//
IAS_Detector_Type internal_Detector_Type;
// No EHS declaration as there are
// no architectural interactions and
// no semi-synchronous interactions.
//---------- DEFINING CONSTRUCTOR --------------
Detector_Type() {
    defineBehavEquations();
}
//--------------- DEFINING BEHAVIOR --------------
void defineBehavEquations() {
    Detector_Off = new BehavioralEquationInterface() {
        public void behavEqCall() {
            _Detector_Off();
        }
        private void _Detector_Off() {
            try {
                turned_engine_on.receive();
            } catch(SyncException e) {} 
            nextBehavEq = Detector_On;
            actualPars = null;
        }
    } // end of behavioral equation Detector_Off
    Detector_On = new BehavioralEquationInterface() {
        public void behavEqCall() {
            _Detector_On();
        }
        private void _Detector_On() {
            switch (ElemMeth.choice(
                    new ChAct[] {
                        new ChAct(true, null),
                        new ChAct(true, turned_engine_off)
                    }) // Choice body :
            { 
                case 0:
                    internal_Detector_Type.measure_speed();
                    try {
                        signal_speed.send();
                    } catch(SyncException e) {} 
                    nextBehavEq = Detector_On;
                    actualPars = null;
                    break;
                case 1:
                    try {
                        turned_engine_off.receive();
                    } catch(SyncException e) {} 
                    nextBehavEq = Detector_Off;
                    actualPars = null;
                    break;
                default:
                    nextBehavEq = null; // STOP
                    actualPars = null;
                    break;
            }
        }
    } // end of behavioral equation Detector_On
//---------- RUNNING ELEMENT [thread] --------//-
Thread th_Detector_Type = null;
public void start() {
    th_Detector_Type = new Thread(this).start();
}
public void join() throws InterruptedException {
    th_Detector_Type.join();
}
public void run() throws InterruptedException {
    internal_Detector_Type = new IAS_Detector_Type();
    nextBehavEq = Detector_Off;
    actualPars = null;
}
while (nextBehavEq != null)
    nextBehavEq.behavEqCall();
}
}

The only internal action measure_speed is translated
into a method of the following stub class:

class IAS_Detector_Type {
    IAS_Detector_Type() {
        // FILL IN THE CONSTRUCTOR BODY IF NEEDED
    }
    void measure_speed() {
        // FILL IN THE METHOD BODY
    }
}

Section ARCHITOPOLY of the PADL description
is translated into a thread class with a core section called
BUILDING ARCHITECTURE:

class Cruise_Control implements RunnableArchi {
    //---------- DECLARING RUNNABLE ELEMENTS --------
    Sensor_Type S;
    Controller_Type C;
    Detector_Type D;
    Actuator_Type A;
    //--- DECLARING ARCHITECTURAL INTERACTIONS ----
    // No architectural interactions are declared
    // ATTACHING LOCAL INTERACTIONS:
    ArchiMeth.attach(S.turn_engine_on,
                    C.turned_engine_on);
    ArchiMeth.attach(S.turn_engine_on,
                    D.turned_engine_off);
    ArchiMeth.attach(S.turn_engine_off,
                    C.turned_engine_off);
    ArchiMeth.attach(S.turn_engine_off,
                    D.turned_engine_on);
    ArchiMeth.attach(S.press_accelerator,
                    C.pressed_accelerator);
    ArchiMeth.attach(S.press_brake,
                    C.pressed_brake);
    ArchiMeth.attach(S.press_on,
                    C.pressed_on);
    ArchiMeth.attach(S.press_off,
                    C.pressed_off);
    ArchiMeth.attach(S.press_resume,
                    C.pressed_resume);
    ArchiMeth.attach(C.trigger_enable,
                    A.triggered_enable);
    ArchiMeth.attach(C.trigger_record,
                    A.triggered_record);
    ArchiMeth.attach(C.trigger_disable,
                    A.triggered_disable);
    ArchiMeth.attach(D.signal_speed,
                    A.signalled_speed);
}
//---------- BUILDING ARCHITECTURE ---------
buildArchITopology();

//---------- RUNNING ARCHITECTURE ---------
Thread th_Cruise_Control = null;
public void start() {
    th_Cruise_Control = new Thread(this).start();
}
AETs Sensor_Type, Controller_Type, and Actuator_Type are translated into thread classes in a way very similar to Detector_Type.

The only intervention of the software developer has to do with the stubs IAS_Detector_Type and IAS_Actuator_Type for the management of the internal actions measure_speed, record_speed, enable_speed_ctrl, disable_speed_ctrl, and adjust_throttle. If the guidelines provided in [9] are observed while filling in those stubs, then the code generated by PADL2Java conforms to the architectural description of Sect. 2.1 – from which it has been synthesized – and hence it turns out to be deadlock free as requested at the beginning of Sect. 2.1.

6. Integrating PADL2Java in TwoTowers

In order to support property prediction and preservation at the software architecture level of design, we have extended the open-source software tool TwoTowers [7] with the capability of generating software. TwoTowers automates the functional verification, security analysis, and performance evaluation of systems modeled in Æmilia [1], a variant of PADL in which action durations can be expressed too.

As shown in Fig. 4, TwoTowers is equipped with a graphical user interface through which the user can invoke several routines by means of suitable menus. The graphical user interface takes care of the integrated management of the various file types needed by the different routines. These routines belong to the Æmilia compiler, the equivalence verifier, the model checker, the security analyzer, and the performance evaluator.

The novelty of version 6.0 is the integration of the PADL2Java code generator. Although PADL2Java has been initially designed for PADL, it can also synthesize Java code out of a correct Æmilia description – differences in the action structure of the two languages are overridden at parsing time. A package synthesized with PADL2Java can be augmented with a class containing method main() or a JApplet-derived class, thus resulting in a Java program or a Java applet, respectively.

7. Conclusion

In this paper we have presented the code generation tool PADL2Java, together with its integration in the modeling and verification tool TwoTowers. Such an integration realizes an architecture-centric approach to software system design that goes from specification down to code in a formal and automated manner. This has been used in case studies ranging from systems for cruise control, audio processing, and video animation to leader election algorithms.

Comparisons with related approaches to code generation from architectural descriptions can be found in [8, 9, 10]. As far as implementation issues are concerned, it has turned out to be quite helpful the adoption of generative techniques for the rapid construction of visitor class hierarchies, used for inspecting the internal structure of architectural descriptions and synthesizing code from it. With regard to stubs, we have found them more appropriate than abstract classes because they allow the generated code to be compiled, including invocations of stub methods as they are concrete. In order to effectively remind the software developer to fill in the stubs, it would suffice to cause PADL2Java to introduce a statement in the definition of each stub method, which prints out an explicative message whenever an empty stub method is invoked at run time.

Concerning future work, we would like to extend our toolset with software model-checking tools, like Java...
PathFinder [26], and to define specific rules for static analysis tools, like TPTP [17]. The reason is that the preservation at the code level of the properties proved at the architectural level is guaranteed only if – the underlying platform is correct and – the guidelines prescribed by the second transformation model are followed when filling in the stubs for internal actions and interaction exceptions (see [9] for more details). Having a software model-checker available within TwoTowers would permit the verification of the overall system after possible manual interventions of the software developer, while customized static analysis tools may be exploited for guiding the interventions themselves.

References