Automatic Code Generation: From Process Algebraic Architectural Descriptions to Multithreaded Java Programs

Edoardo Bontà

28 April 2008

Introduction

- The increasing importance of **multimedia** and **real-time applications**, together with the need of fully exploiting the features of **multicore/multiprocessor systems**, ever more frequently requires the **critical activity of developing large portions of concurrent software**.
- In general, as observed in the last decades, the provision of suitable **notations for formal or semi-formal system modeling** is a good solution for **dealing with the increasing complexity and size** of modern software systems.
- Among the formal notations proposed in the literature, the **architectural description languages based on process algebra** seem to be one of the most promising approaches for supporting the implementation of concurrent software systems.

Implementing Concurrent Software Systems from Architectural Models

SOURCE DOMAIN

Architectural Description Languages + Process Algebra

TARGET DOMAIN

Object-Oriented Programming Languages + Multithreading

Problem: *how do we guarantee that a program is consistent with its model?*

models

Process Algebraic Architectural Descriptions

code

Multithreaded Object-Oriented Programs

PROPOSAL

✓ **PADL**

✓ **Java**

✓ **Automatic Code Generation**

in a "correct-by-construction" way

Structure of a PADL Textual Description

ARCHI_TYPE	AT name and initialized formal data parameters
ARCHI_BEHAVIOR	
…	
ARCHI_ELEM_TYPE	AET name and formal data parameters
BEHAVIOR	sequence of process algebraic defining equations built from stop, action prefix, choice, and recursion
INPUT_INTERACTIONS	input synchronous/semi-synchronous/asynchronous uni/and/or-interactions
OUTPUT_INTERACTIONS	output synchronous/semi-synchronous/asynchronous uni/and/or-interactions
…	
ARCHI_TOPOLOGY	
ARCHI_ELEM_INSTANCES	AEI names and actual data parameters
ARCHI_INTERACTIONS	architectural-level AEI interactions
ARCHI_ATTACHMENTS	attachments between AEI local interactions
[BEHAV_MODIFICATIONS]	names of actions to be hidden, restricted, or changed
END	

Producing Code: A Model-Driven Approach

↓ **Source domain**: models expressed in the process algebraic architectural descriptions language PADL.

↓ **Target domain**: Java code.

→ **Model-to-code transformation**

- Since source and target domains are disjoint, an exogenous transformation must be adopted.

✓ The provision at the target side of a library of software components inspired by the main architectural concepts can **reduce the gap between code and model**.

- This makes the transformation a "semi-exogenous" one, by reducing the effort for the implementation and the redundancy of the produced code.

✓ Several benefits derive from the **automatic code generation**.

Automatic Code Generation: Benefits

- ✓ **Correctness-by-construction**
  - The approach of manually writing code from models is tedious and error prone. Once a **correct** and **complete model-to-code mapping** has been established, instead, the **automatic code generation** approach **avoids any translation problem**.
- ✓ **Concurrency management at high level of abstraction**
  - A **communication model** is provided at the **architectural level**, thus **developers are saved from reasoning about complicated combination of synchronization techniques** – e.g., sleep, wakeup, and notify primitives, synchronized methods, etc. The **low level work** for handling multithreading is done by [predefined architecture-inspired software components instantiated and assembled by] the **generated code**.
  - Suitable **techniques and tools for assessing the correctness** of communications can be applied **at the architectural description level**.
- ✓ **Cost saving in terms of time and resources**

Three-Phases Approach for Generating Code

1. Thread Coordination Management
2. Thread Behavior Generation
3. Monitors Synthesis

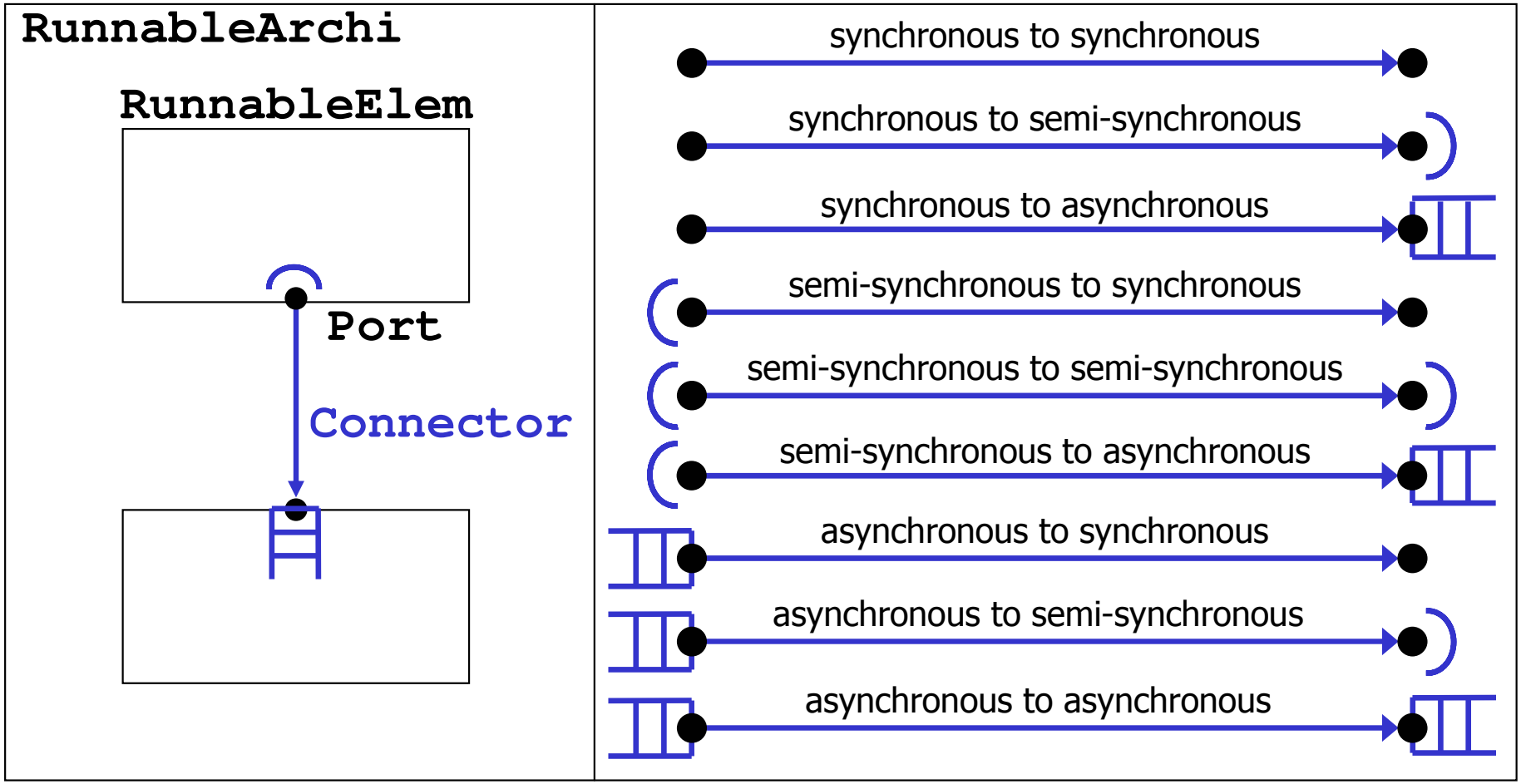
Thread Coordination Management: The Java Package **`Sync`**

- A Java package called **`Sync` takes care of the details of the thread synchronization** in a way that is transparent to the developer.
- The components of the package are inspired by the main **architectural concepts**: architectures, components, connectors, ports.
- The package is **consistent with the communication model of PADL**.

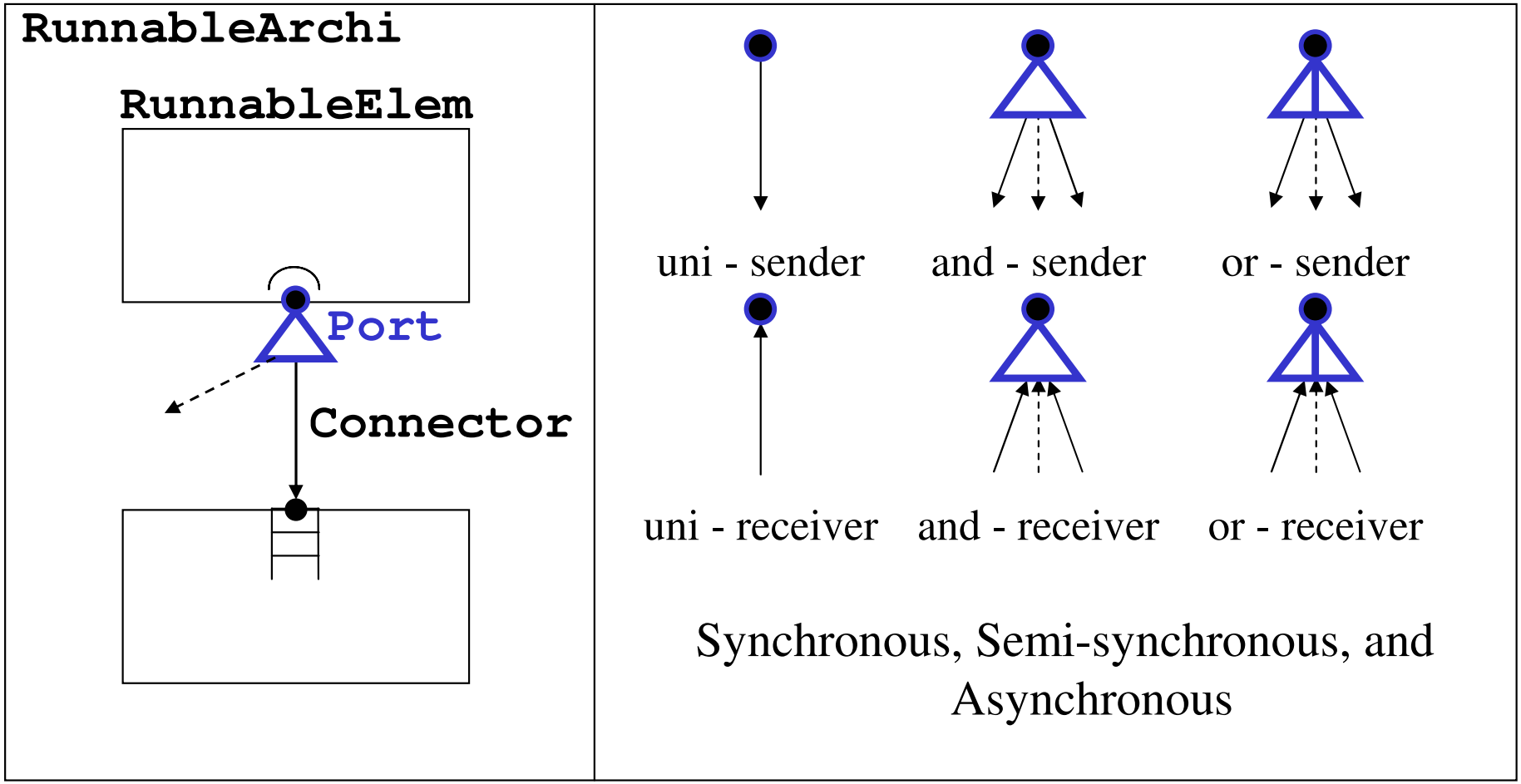
Thread Coordination Management: Generating Code

1. **Provide an architectural description** in PADL of the multithreaded program, which **specifies at a high level of abstraction** the **program topology** in terms of thread **instances**, their **interactions**, and their behavior (the latter will be useful in the second phase of the approach).
2. **Use a translator** that, from the PADL specification of the multithreaded Java program **automatically generates** (a skeleton of) **the program** based on the Java package.

Java package **Sync**: Layer `Connector`

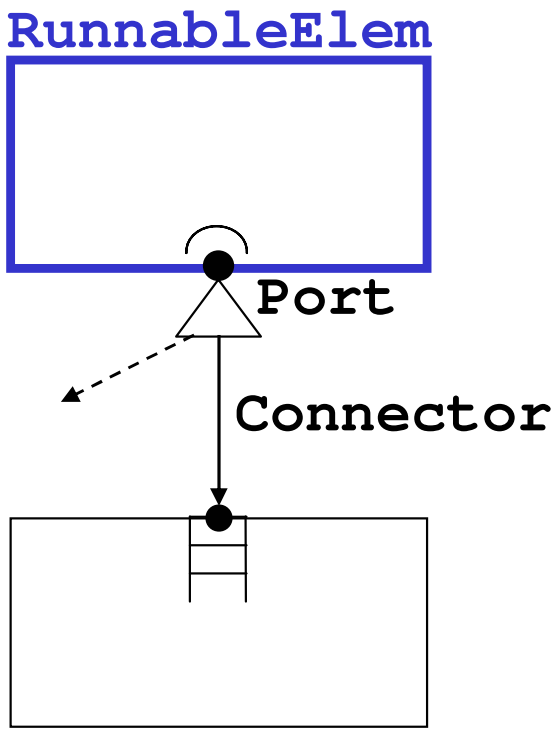


Java package **Sync**: Layer `Port`



Java package **Sync**: Layer `RunnableElem`

RunnableArchi

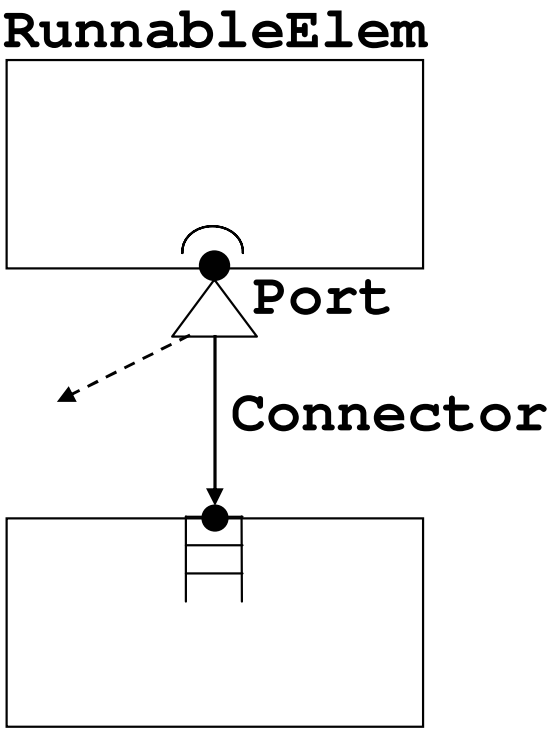



```
public interface RunnableElem
    extends Runnable {…}
```

- Thread classes must implement the interface `RunnableElem`.

Java package **Sync**: Layer `RunnableArchi`

RunnableArchi




```
public interface RunnableArchi
    extends RunnableElem {…}
```

Advantages deriving from inheritance:

- Hierarchy
- Composition
- Concurrency

Generated Code

```
import Sync.*;
```

```
class <Architecture Type Name>
   implements RunnableArchi {
```

Sections:

- DECLARING THREADS (element types)
- DECLARING ARCHITECTURAL INTERACTIONS
- DEFINING CONSTRUCTOR
- BUILDING ARCHITECTURE
- RUNNING ARCHITECTURE

```
}
```

```
class <Element Type Name>
   implements RunnableElem {
```

Sections:

- DEFINING CONSTRUCTOR
- /* TODO: DEFINING BEHAVIOR */
- INSTANTIATING INPUT INTERACTIONS
- INSTANTIATING OUTPUT INTERACTIONS

```
}
```

Second Phase of the Approach

1. Thread Coordination Management
2. Thread Behavior Generation
3. Monitors Synthesis

Thread Behavior Generation
Property Preservation

- **Translate the algebraically-specified behavior of each AET into the implementation code of a thread class.**
- ***How to translate the specification of the thread behavior into Java code in a way that preserves the properties proved at the architectural level?***
- Only a **partial translation based on stubs** is possible, with the **property preservation depending on the way in which the stubs are filled in**.
- **Extending our code generator** to provide support for the thread behavior translation and the stub generation.

Generated Code

```
import Sync.*;
```

```
class <Architecture Type Name>
   implements RunnableArchi {
```

Sections:

- DECLARING THREADS (element types)
- DECLARING ARCHITECTURAL INTERACTIONS
- DEFINING CONSTRUCTOR
- BUILDING ARCHITECTURE
- RUNNING ARCHITECTURE

```
}
```

```
class <Element Type Name>
   implements RunnableElem {
```

Sections:

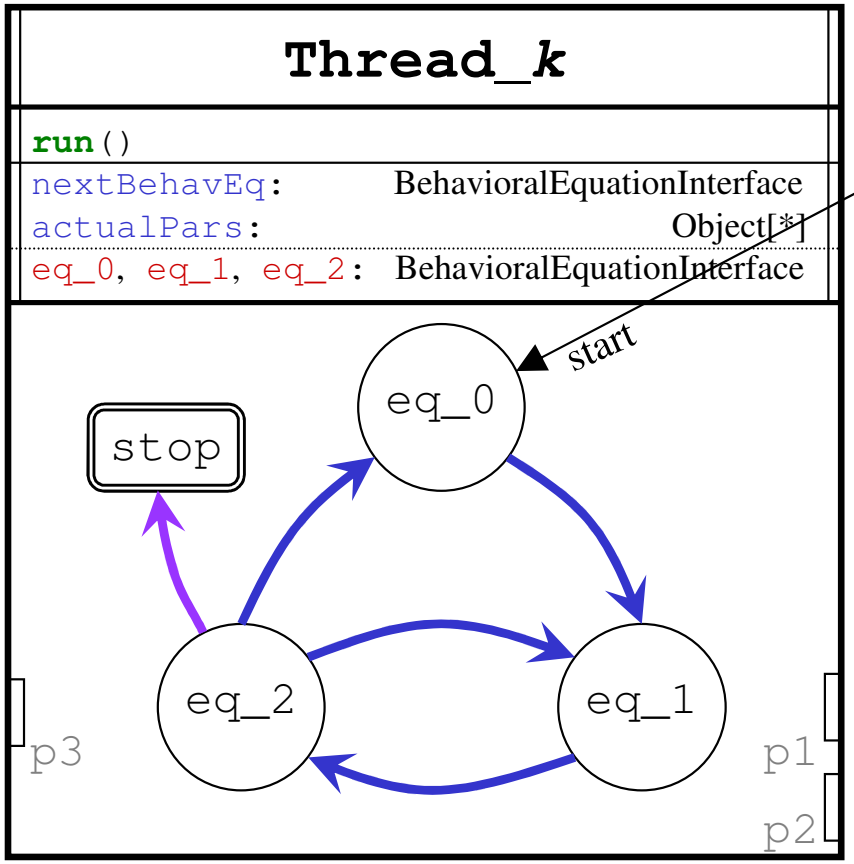
- DEFINING CONSTRUCTOR
- DEFINING BEHAVIOR
- INSTANTIATING INPUT INTERACTIONS
- INSTANTIATING OUTPUT INTERACTIONS

```
}
```

Translating the Thread Behavior

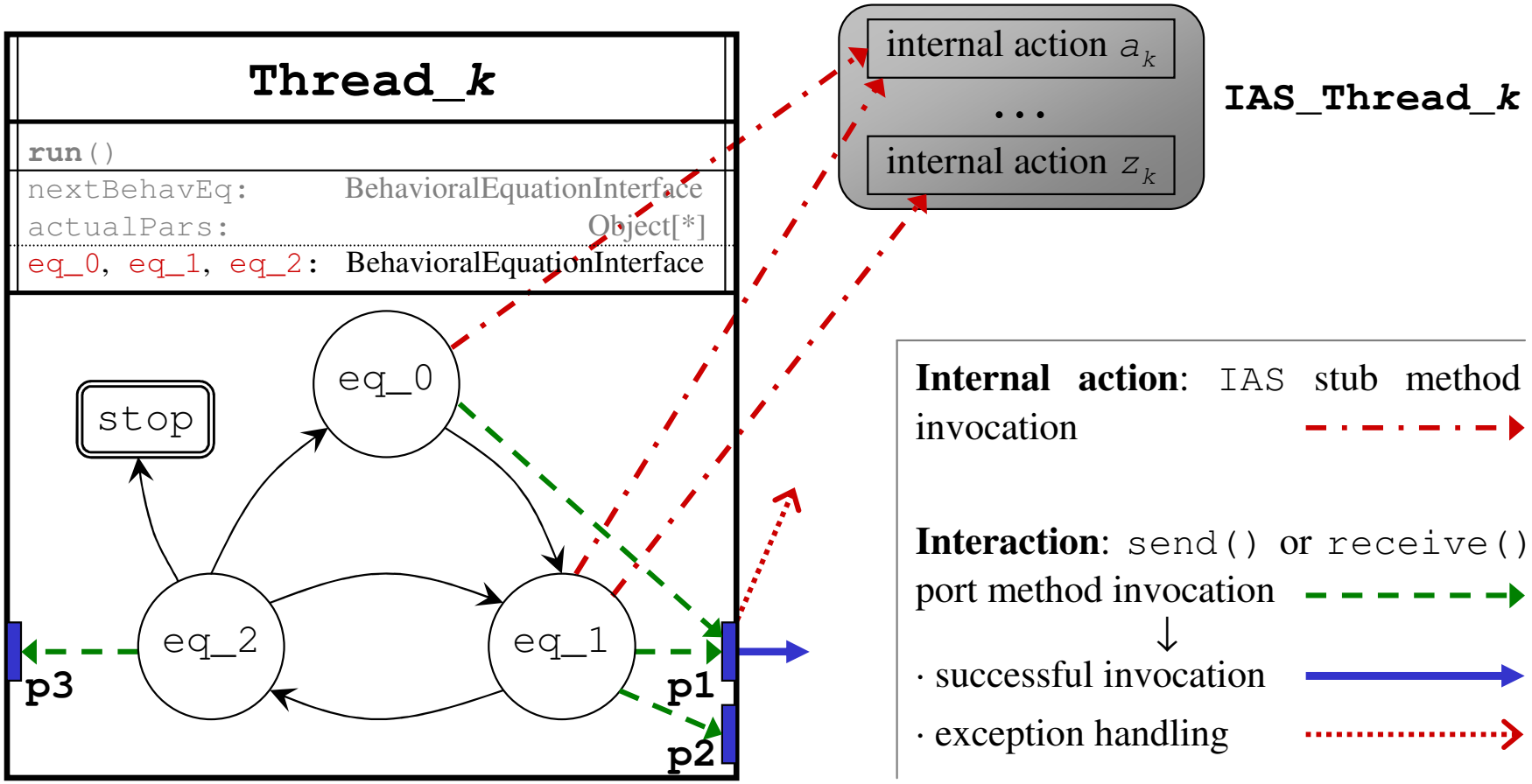
- **A control-flow structure** is generated on the basis of the **behavioral design pattern "State"**.
- **Method `run()`** of the `RunnableElem`-implementing class **guides the execution of the behavioral equations** (i.e. the process algebraic defining equations that specify the behavior of an AET).
- **Each behavioral equation** of an AET specification is **translated into an instance of a behavioral class** implementing the interface `BehavioralEquationInterface`.
- The **interface method `behavEqCall()` of each behavioral class is implemented by proceeding by induction on the algebraic structure of the corresponding behavioral equation**.

Behavioral Invocations & Process Term Stop

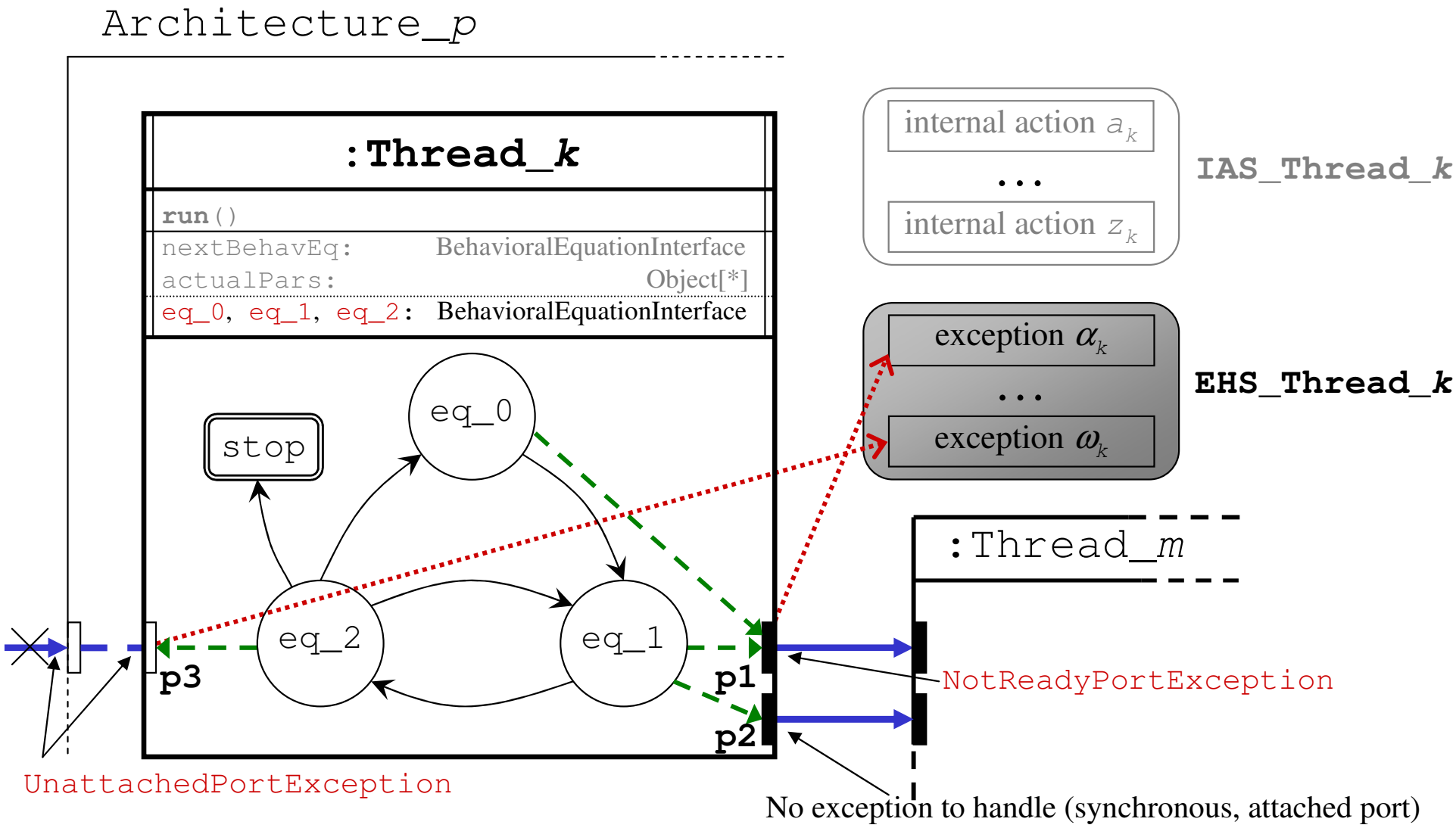



```
public void run() {
  <IAS instantiation>
  <EHS instantiation>
  nextBehavEq = eq_0;
  actualPars = <eq_0 parameters>
  while (nextBehavEq != null);
    nextBehavEq.behavEqCall();
}

class eq_0 ...

class eq_1 …

class eq_2 implements
            BehavioralEquationInterface{
  public void behavEqCall()
  …
  …; nextBehavEq = eq_0;
     actualPars = <eq_0 parameters>;
  …; nextBehavEq = eq_1;
     actualPars = <eq_1 parameters>;
  …; nextBehavEq = null;
}
```

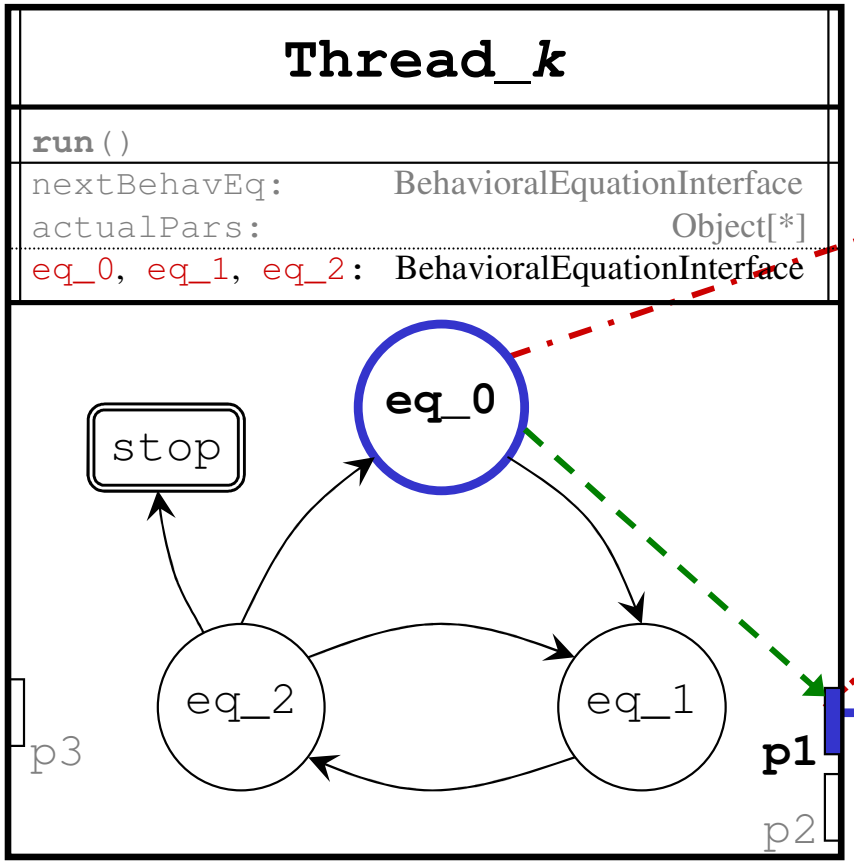
Action Prefix



Interactions – Exceptions



Choice

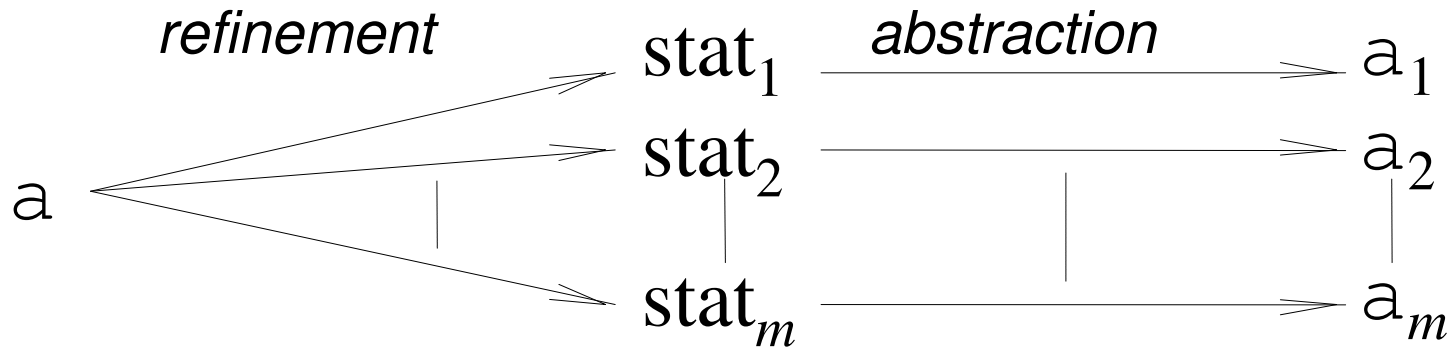



```
public void eq_0() {
  ...
  switch(ElemMeth.choice(<choice list>)) {
    case 0:
      internal_Thread_1.<action a(params a)>;
      ...
      nextBehavEq = eq_1;
      actualPars = <eq_1 parameters>
      break;
    case 1:
      try {
        p1.send(<params p1>);
      } catch(SemisyncPortNotReadyException e) {
        except_Thread_1.<exception α(params p1)>;
      }
      ...
      nextBehavEq = eq_1;
      actualPars = <eq_1 parameters>;
      break;
    default:
      nextBehavEq = null;
  }
}
```

Preserving Architectural Properties

- PADL is equipped with a **component-oriented technique** based on **equivalence checking** for verifying **the freedom from architectural mismatches** [AB 2005].
- *Are such properties preserved when going to the code level?*
- Three aspects:
  1. Code generated for the thread management: ✓ Ok
  2. Code generated from the behavioral equations: ✓ Ok
  3. Code provided for filling in the stubs: don't know...

Property Preservation Result



Theorem. *Let* T *be the process algebraic description of the behavior of a thread and let* a *be an internal action occurring in* T. *Let* a_1, a_2, ..., 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* a._ *with* $a_1.a_2. \ldots .a_m._$. *Let* H *be the set of internal actions occurring in* T *or* T′. *Whenever* T *satisfies* $\mathcal{P}$ *and* $a.stop \setminus H \simeq_B a_1.a_2. \ldots .a_m.stop \setminus H$, *then* T′ *satisfies* $\mathcal{P}$ *as well.*

Guidelines for Filling in the Stubs (1/2)

- 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`**.

Guidelines for Filling in the Stubs (2/2)

- 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.

Third Phase of the Approach

1. Thread Coordination Management
2. Thread Behavior Generation
3. Monitors Synthesis

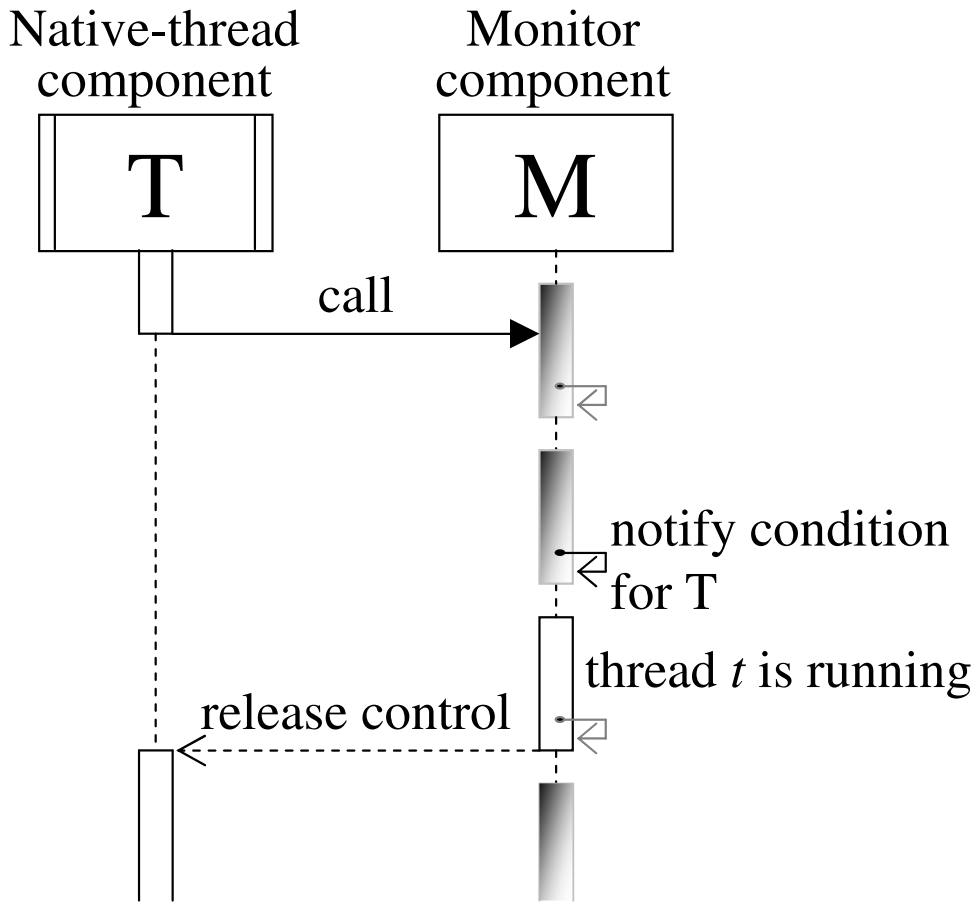
Monitor Synthesis

- **Translate into monitors** (instead of threads) **the algebraically-specified behavior of software components that satisfy a given set of constraints.**
- w.r.t. the previous phase, where only the automatic generation of thread classes is taken into account, **the performance of the generated code may be improved** thanks to the **synthesis of monitors** as they **would reduce the thread context switch frequency**.

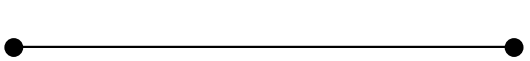
Application Steps for Synthesizing Monitors

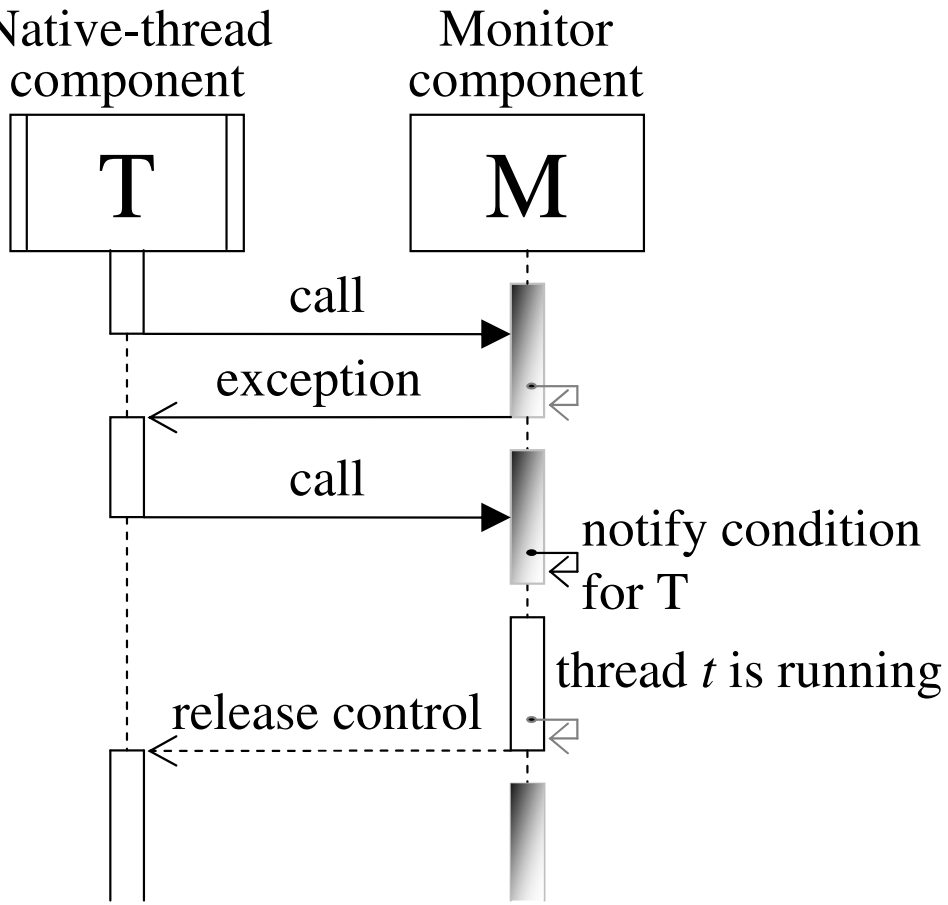
1. **Check the constraints** that guarantee the derivability of a monitor from a process algebraic component description.
2. **Rewrite the process algebraic component description** into a form called ***monitor normal form*** – from which it is easy to proceed with the third step.
3. **Synthesize a Java monitor** from the monitor normal form.

Thread-Monitor Interaction Models

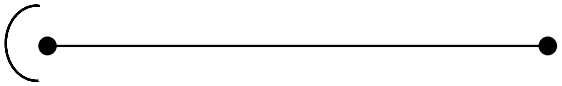


Synchronous model



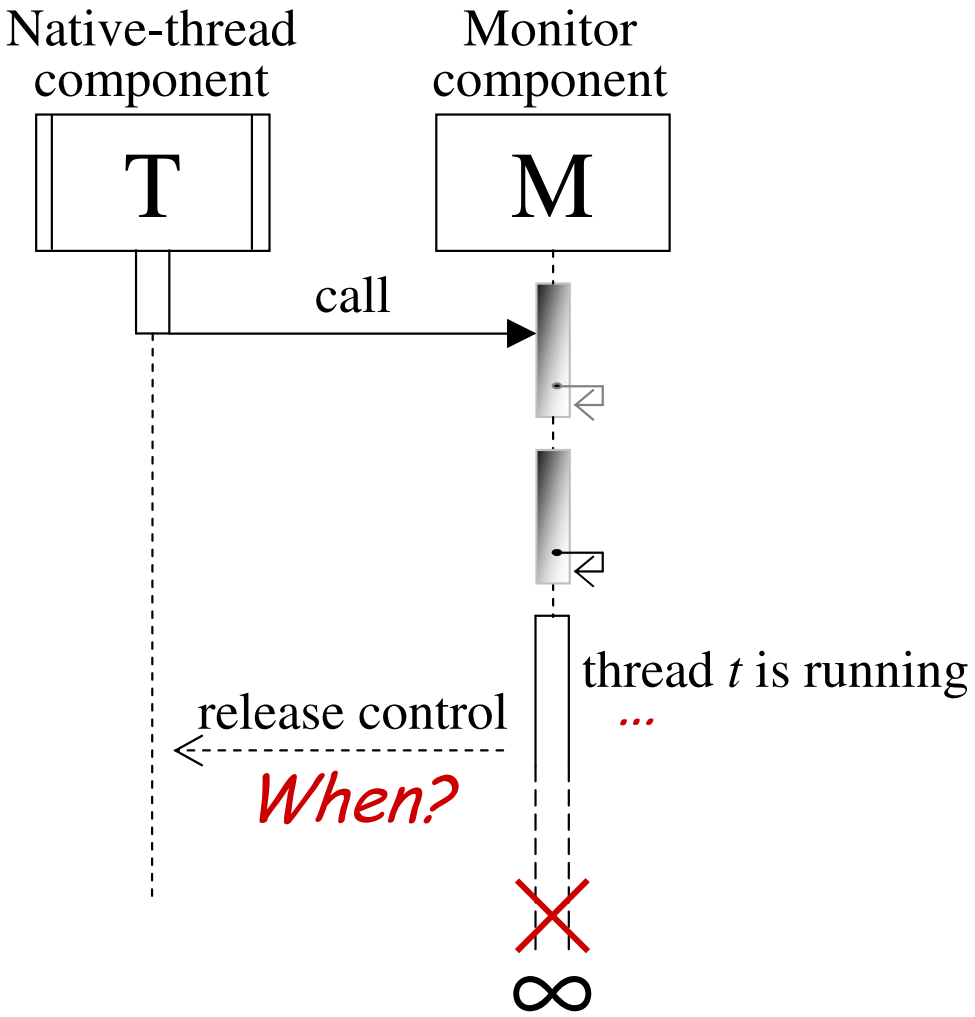


Semi-synchronous model



Monitor Constraints (1/4)

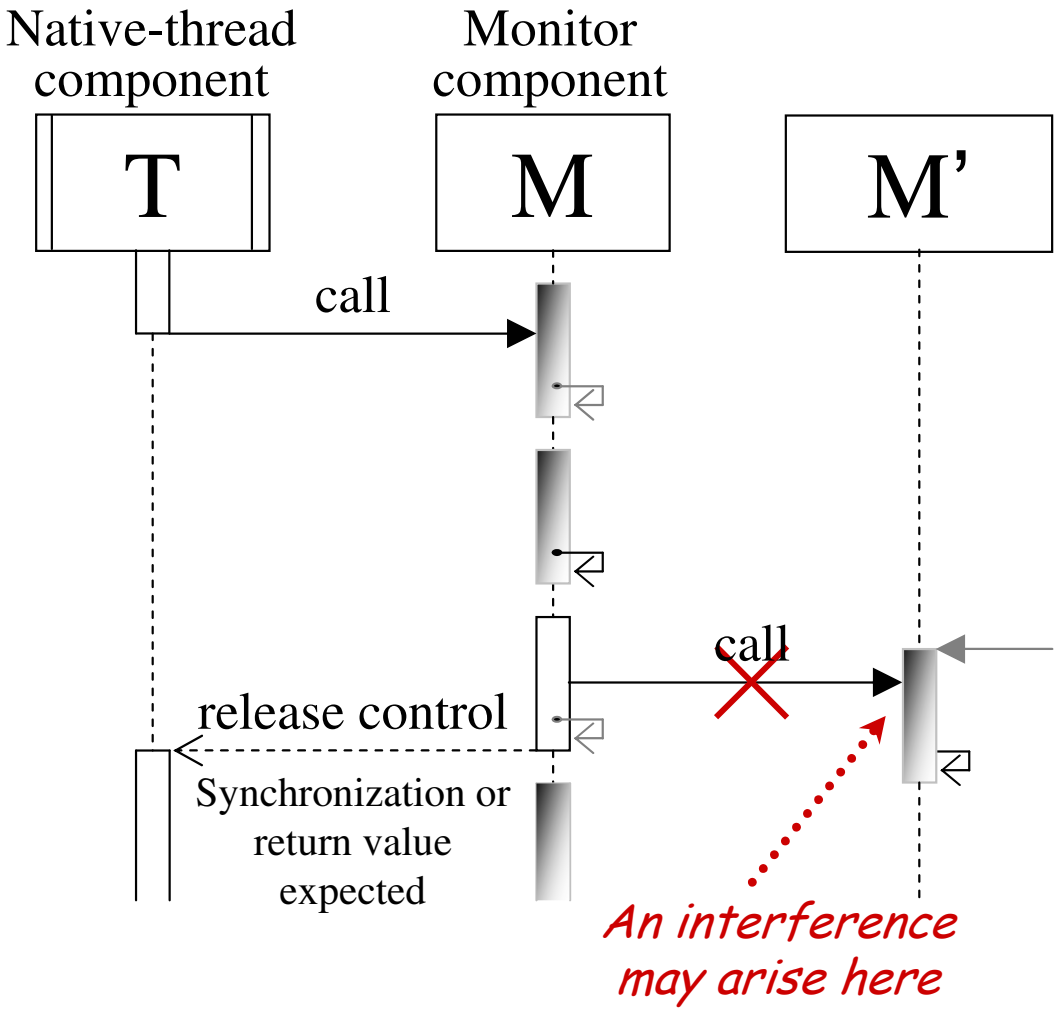
No Cycles of Internal Actions



- Thread *t* taking the control of a monitor (M) would have to run inside it only for a finite amount of time.
- In the process algebraic description of a candidate monitor type, the maximum number of consecutive internal actions that can be performed must be finite.
- This constraint can be checked by verifying the absence of cycles of internal actions in the process algebraic description of the candidate monitor type.

Monitor Constraints (2/4)

No Attached Monitor Type Instances

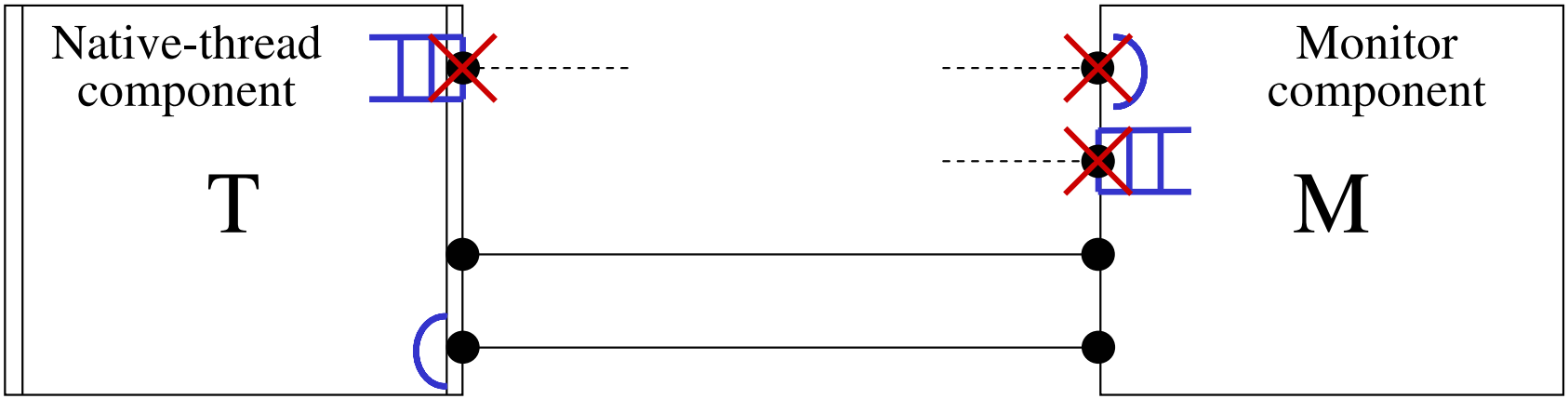


- Thread *t* taking the control of a monitor (M) should not move to another monitor (M') before leaving the monitor in which it is running. This constraint avoids monitors interferences on the thread of control *t*.
- The constraint can trivially be verified by examining the attachments in the topological section of the process algebraic architectural description in which instances of the monitor type are involved.

Monitor Constraints (3/4)

No Non-Synchronous Interactions

- Since a monitor is a passive entity, it cannot perform active-control interactions. A monitor can only passively communicate with a thread in a synchronous way and a thread can communicate only synchronously or semi-synchronously with a monitor.



- This can trivially be verified through the qualifiers expressing the synchronicity of the communications in which the instances of the candidate monitor type are involved.

Monitor Constraints (4/4)

No Non-Disjoint Hybrid Choices

release control

continue

$$\ldots \rightarrow \begin{cases} \text{cond}(G_{\alpha_0}) \rightarrow \alpha_0.P_{\alpha_0} \\ \ldots \\ \text{cond}(G_{\alpha_n}) \rightarrow \alpha_n.P_{\alpha_n} \\ \text{cond}(G_{\beta_0}) \rightarrow \beta_0.P_{\beta_0} \\ \ldots \\ \text{cond}(G_{\beta_m}) \rightarrow \beta_m.P_{\beta_m} \end{cases}$$

$\alpha_0 \ldots \alpha_n$: passive-control interactions
$\beta_0 \ldots \beta_m$: internal actions

$$\left(\bigvee_{i=0}^{n} G_{\alpha_i}\right) \wedge \left(\bigvee_{i=0}^{m} G_{\beta_i}\right) \equiv false$$

Ok

- A hybrid choice is a choice between a non-empty set of interactions and a non-empty set of internal actions.
- A choice between a passive-control interaction (release control) and a internal action (continue) would make it impossible to decide whether the currently running thread has completed its task or not, unless the two actions are preceded by disjoint conditions.

Monitor Normal Form

Once **all the constraints are satisfied** by the process algebraic description of a candidate monitor type, it is possible to proceed to the **transformation** of the description itself **into monitor normal form**.

1. **Rewriting Complex Choices**
2. **Splitting Defining Equations**
3. **Building the Interacting Choice Equation**
4. **Building Setting Equations**
5. **Rearranging the Interacting Choice Equation**

Correctness of the Transformation into Monitor Normal Form

Theorem. *Let M be the process algebraic description of a monitor type and let M′ be the process algebraic description of the monitor normal form obtained by applying to M the syntactic transformation. Then the LTS underlying M′ is isomorphic to the LTS underlying M.*

This theorem can be proved by showing for each of the five steps of the transformation taken in isolation that the labeled transition system underlying the input process algebraic description of the step is isomorphic to the labeled transition system underlying the output process algebraic description of the step.

Monitor Implementation

- **Generating the Core Monitor Class**
  1. Translating Internal Actions into Stub Class Methods
  2. Declaring IAS Stub and Synthesizing the Monitor Class Constructor
  3. Translating Setting and Internal Equations
  4. Translating the Interacting Choice Equation
  5. Synthesizing the Starting Method
- **Generating the Monitor Wrapper Class**
  - the interface `RunnableElem` is implemented together with as many interfaces `Port` as there are synchronized methods in the core monitor class - through the instantiation of anonymous classes that use a single core monitor instance.

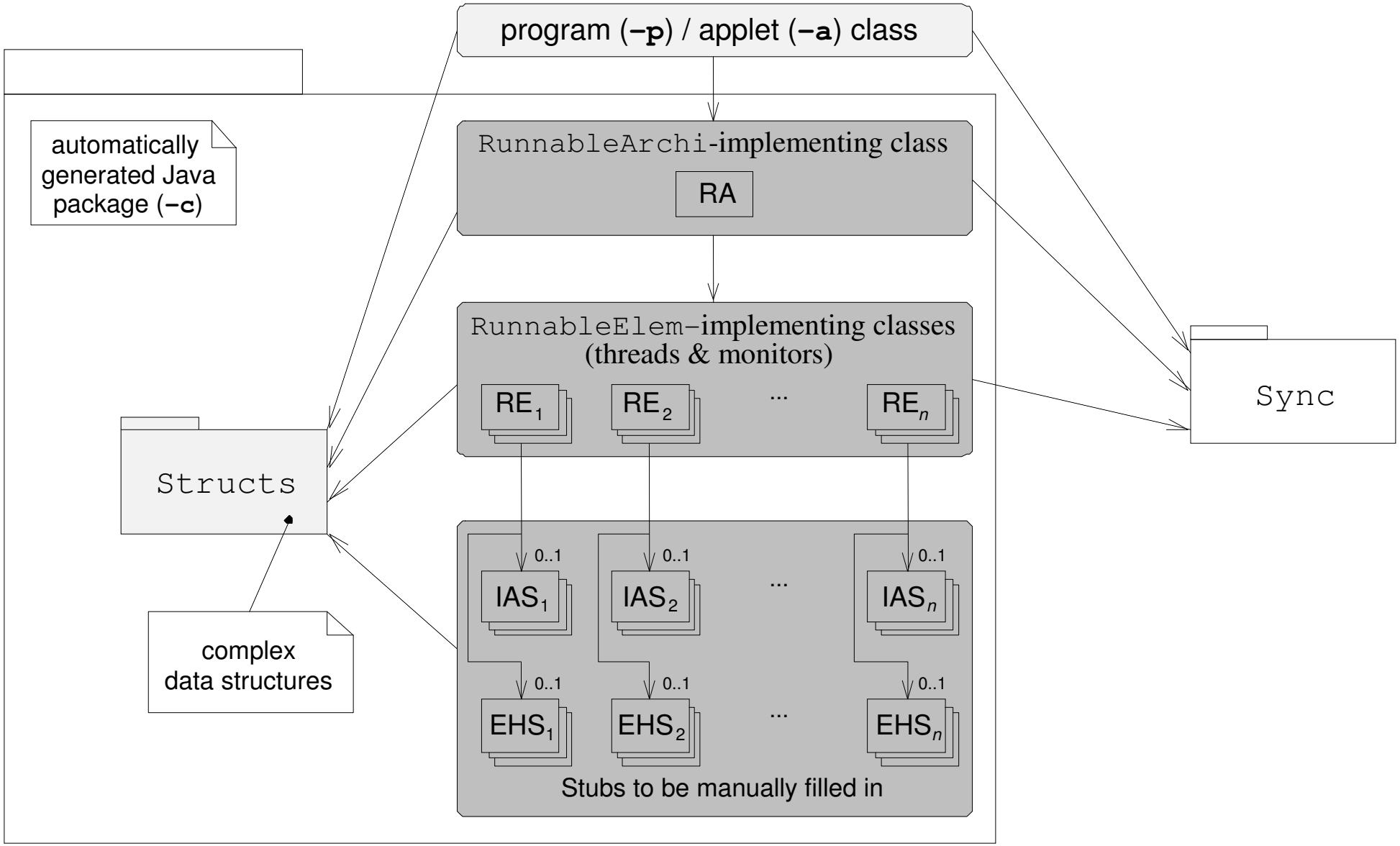
The Translator PADL2Java

- The three-phases approach has been implemented in a **translator** called **PADL2Java**.
- PADL2Java is **composed of a set of Java classes** created by the **parser generator JavaCC**, plus **other classes based on the pattern "Visitor"** for the **analysis**, **transformation**, and **code generation** of a **PADL description**.
- Command line application:

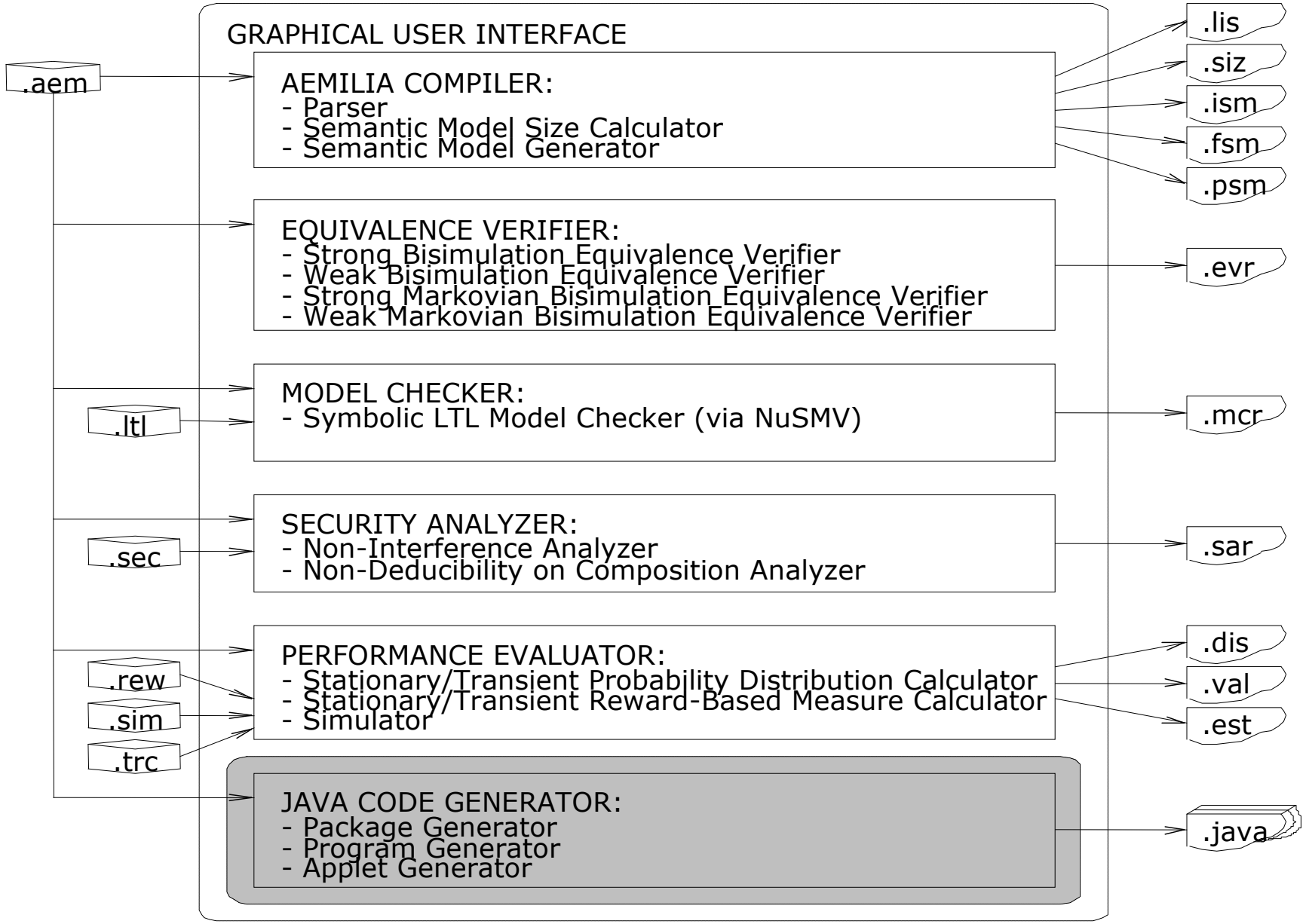
  PADL2Java [**-c**] [**-p <***program***>**.java] [**-a <***applet***>**.java] *source*.**padl**

  – option **`-c`**: classes only
  – option **`-p`**: program (a further class with the function `main()` is generated)
  – option **`-a`**: applet (a furter JApplet-deriving class is generated)

File Structure of the Generated Code



Integration of PADL2Java in TwoTowers



Case Studies

Three case studies have been investigated:

- **Audio Processing System (running example)**
- **Video Animation Repainting System**
- **Leader Election Algorithm**

Conclusion (1/2)

Contribution

- Proposed an **approach for automatically generating multithreaded Java programs from process algebraic architectural descriptions** of concurrent software systems.
- **Implemented** the approach in **the translator PADL2Java**.
- **Integrated PADL2Java in** the architecture-centric verification tool **TwoTowers**.
- **Enhanced the expressiveness of** the architectural description language **PADL**
  - Introduced non-synchronous interactions (semi-synchronous and asynchronous)
  - Introduced the generic object data type (`object(<`*type id*`>) obj`)

Conclusion (2/2)

Related Work and Future Research

- Most important related work: **ArchJava**.
- Future research/extensions:
  - Possible **integration with ArchJava to take advantage of their complementary strengths**.
  - Possible **integration with Bandera for software model-checking**.
  - Definition of **specific rules** for **static analysis tools** (e.g. Eclipse TPTP) for **guiding the intervention of the software developer** when filling in the stubs for internal actions and exceptions.
  - Further **investigations on the monitor constraints**, in particular **with respect to specific contexts**.