Protocol analysis in distributed applications with model checking of timed automata

Cand. Inform. Piotr Serwa
serwap@gmail.com

Thesis
at
Forschungszentrum Informatik
Universität Karlsruhe (TH)
Forschungsbereich Softwaretechnik
Prof. Dr. Walter F. Tichy
Dipl. Inform. Marc Schanne
"Program testing can be used to show the presence of bugs, but never to show their absence!"
E. Dijkstra

“Every protocol should be considered to be incorrect until the opposite is proven.”
E. Holzmann

"Beware of bugs in the above code; I have only proved it correct, not tried it."
D. Knuth
Hereby I declare that I have done this work autonomously. All used sources are fully quoted.

Piotr Serwa

Karlsruhe, 2006-08-31
Abstract

Within this document, we propose an approach for efficient design analysis of distributed java applications, having a purpose of efficient design and error finding in early stages of development process. In this approach, a formal model is extracted from java source code with support of code annotations supported by developer and then checked for errors. The tool specification and proof-of-concept has been provided, and it has been validated on an application example. The example has proved the approach and the tool to be an efficient and lightweight solution.

In mission-critical systems, it is important to keep the software errors down to minimum. For coding/development errors there are various efficient and frequently used techniques like compiler checks, debugging, assertions, coding guidelines checks, unit testing, which enable to detect errors soon after they occur.

However, for finding errors in design, there is either integration testing, simulation or design verification. The main problem with integration testing is that it occurs at the very late stage of development cycle, where removing of errors is very expensive. Simulation can be executed earlier, but it requires to write much simulation wrappers (like for unit testing), which causes major difficulties for complex distributed applications. Moreover, the coverage of both methods is limited.

Formal methods can be applied to verify the design at early stages of the lifecycle. Moreover, they are very efficient in finding protocol errors. However, the problem is that in most projects, even these mission-critical ones, the design is not complete/precise enough, or it is not reflecting the source code (because of synchronization costs). Moreover, usage of model checking of design requires formalization of the design specification and of requirements – this requires experts in formal methods. As a result, formal model checking failed to win popularity in industry.

To address this issue, an approach has been proposed – it is to extract the formal model automatically from the source code (with a model extraction tool) and then verify this formal model against properties. This approach has still problems: the extraction tool does not know which information is relevant and to which degree the abstraction should be done. As a result, the tools doing automatic model extraction suffer from so-called “state-explosion” and are not scalable.

As a result, we propose here a new approach: a semi-automatic model extraction, addressing the mentioned problems. It is the developer who selects which parts of code are relevant for model checking, and selects how to abstract them. The developer needs simply to annotate Java statements with code comments. Concerning the resulting formal model, a graphical formal language is used (timed automata). A formal model checking tool, UPPAAL, checks the model against properties, but also enables to visually simulate the model, though a modern GUI.
The main advantages of the approach are that the work can be done by developers themselves, the model (thanks to annotations within the source code) can be easily kept synchronized with the code, the output model is simple and defines only aspects/subsystems that are under the analysis. Moreover, due to the simplicity of output and usage of graphical modeling language, the developer can also simulate the model to find further errors as well as to see and understand the error sequence causing the violation of a property.

Within this diploma thesis, we specify the architecture and design of the tool following the above approach. The core of this document is the use-case driven specification of functionalities to be provided.

The proof-of-concept implementation of the tool has been done; within this document we just summarize the implementation status. Moreover, as a validation of the approach and of the specification, we present an existing application model (UPPAL train-gate example), and we demonstrate how the errors in the source code are found through property checks and model simulation.

In conclusion, the specified approach and tool specification proved to be a viable and efficient solution to detect errors in early development stages in Java projects.
Kurzfassung (Deutsch)


Die Arbeit verwendet ein existierendes Test-Szenario des EU-Forschungsprojekts für High Integrity Java Applications (HIJA) und zeigt in einem „Proof-of-Concept“ die Funktionstüchtigkeit des Verfahrens. Bei der Validierung einer Beispielanwendung wird gezeigt, dass das Werkzeug eine effiziente und kompakte Lösung darstellt.

Einführung


Als Machbarkeitsstudie der vorgestellten Lösung wird schließlich die Validierung eines vorhandenen Anwendungsmodells (UPPAAL Train-Gate-Beispiel) vorgestellt. Wir demonstrieren, wie die Fehler im Quellprogramm durch Eigenschaften/Anforderungen in Check- und Modellsimulation gefunden werden.

Als Schlussfolgerung wird eine allgemeine Vorgehensweise definiert und das neue Analyseverfahren als mächtige und effiziente Lösung bei der Identifizierung von Fehlern in einer frühen Entwicklungsphasen in Java Projekten vorgeschlagen.

### Analyse


Das dargestellte Verfahren kann angewendet werden, um das Design in einer sehr frühen Phase des Entwicklungsprozesses zu überprüfen. Das Quellprogramm muss noch nicht existieren. Es erfordert zwar die Verfügbarkeit eines vollständigen Designs, kann aber besonders beim Aufspüren von Kommunikationslogikfehlern nützlich sein.

Da in vielen Projekten das Design nicht komplett oder exakt genug ist, um ein formales Modell zu erzeugen konnte die formale Modellüberprüfung bisher noch keine größere Popularität in der Industrie gewinnen. Solcher Verfahren werden nur in der Avionik und Luftfahrt allgemein verwendet. Ein Beispiel des Gebrauches in anderen Industrien ist die formale Überprüfung eines Sicherheitsprotokolls mit PROFIsafe [R19].
Ein alternatives Verfahren, das die Schwierigkeiten bei vom Design abgeleiteten Modellen umgehen soll, ist Erzeugung des formalen Modells aus dem Quellprogramm, mit automatischer oder halbautomatischer Extraktion.

Bei der automatischen Modellextraktion ist die Hauptschwierigkeit eine logische Abstraktionslücke und der semantische Abstand zwischen einem nicht-formalen Software-System, das als Quellprogramm vorliegt und einem formalen Modell für die Analyse. Es ist schwierig diese Lücke mit komplexer Programmanalyse-, Abstraktions- und Transformations-Techniken zu füllen.

Dennoch kann, wie durch diese Diplomarbeit vorgeschlagen, die Modellextraktion vom Entwickler kontrolliert werden. Dies ermöglicht es, das hohe Niveau der Abstraktion zu erreichen und das Modell auf nur relevanten Elementen zu verringern.

**Aktuelle Technologien**


**Konzept**

Als Ergebnis der Analyse werden im Folgenden Bedingungen für das Protokollverifizierungstool (PV-Tool) ermittelt. Das entwickelte Konzept wird eingeführt und Bedingungen für die nachfolgenden Kapitel beschrieben.

1. Das Werkzeug unterstützt Java 2 SE mit Quellcode-Annotationen als Eingabesprache;
2. Das Werkzeug unterstützt Analyse der Programme, wobei Teile des Systems nur als kompilierte Bibliotheken zu Verfügung stehen;
3. Annotationen, mit denen der Entwickler die Modellextraktion steuern kann, sind einfach und verwenden Java Standards für die Syntax;
4. Falls Annotationen oder Teile von Annotationen nicht angegeben sind, liefert das Werkzeug ein intuitives verständliches Standardverhalten;
5. Das Werkzeug überprüft Anforderungen, die vom Entwickler verlangt werden, auf Korrektheit und benachrichtigt den Entwickler im Fall eines Fehlers;
6. Das Werkzeug verwendet UPPAAL Modell/Eigenschaften als Ausgabe für den Model-Checking-Schritt der mit UPPAAL durchgeführt wird;
7. Für den Entwickler ist es leicht das Ausgabemodell zu verstehen und den Fehler-Trace von UPPAAL für die Fehlersuche zu nutzen;
8. Der Fehler-Trace kann dazu verwendet werden, um einen Fehler im Modell auf die potentielle Fehlerstelle im Quellcode des Programms zurück zu verfolgen.
9. Das Werkzeug identifiziert Design-/Implementierungsfehler (kann aber nicht die Korrektheit prüfen).
10. Das Werkzeug muss auch für große Java Projekten schnell sein und gut skalieren.


Wir verwenden als Backend UPPAAAL, dessen GUI Schnittstelle verwendet werden kann, um die extrahierten zeitgesteuerten Automaten grafisch zu simulieren. Außerdem profitieren wir so von der aktiven und regen Weiterentwicklung von UPPAAAL und auch für das PV-Tool ist eine entsprechende Erweiterung denkbar.

**Softwarearchitektur**

Im Detail betrachtet verläuft die Extraktion wie folgt. Für die Quellcode-Zerteilung wird zuerst eine externe Übersetzerbau-Zerteilerbibliothek (Recoder) verwendet. Sie erhält das Java-Quellprogramm (als Text) und erzeugt ein sequentielles Objekt-Modell, das dieses Quellprogramm darstellt für die weitere Analyse. Dank Recoder kann bei der Zerteilung des Java-Quellprogramms auf eine komfortable API zurückgegriffen werden.


Dieses Generator-Modell wird zum abschließenden PV-Modell umgewandelt, das die UPPAAL XML Beschreibung in einer Java-Objekte-Darstellung repräsentiert.

Durch die Serialisierung des PV-Modells mit einer externen Bibliothek (XBean) werden die Eingabeformate für das Model-Checking Tool erzeugt. Abbildung 5-2 zeigt diesen Ablauf schematisch.
Die Hauptsteuerteilsysteme und Metamodell-Teilsysteme innerhalb des PV-Tools werden in Abbildung 5-3 vorgestellt:

1. Syntaxanalyse (Parser): verantwortlich für die Extraktion des intermediären Modells (dem Syntaxanalyse-Modell),
2. Syntaxanalyse-Modell (Parser Model): Klassenhierarchie, um den Programmcode als Sequenz der Vorgänge (Anweisungen) zu formen,
3. Generator: verantwortlich für die Erzeugung des abstrakten Anwendungmodells (dem Generator-Modell),
4. Generator-Modell: Klassenhierarchie um den Ablauf eines verteilten Programms in einer hierarchischen, abstrakten Sicht darzustellen,
5. Abschließendes PV-Modell: Darstellung der UPPAAL-Eingaben als Java-Klassen vor der Serialisierung in XML Dateien.
Use-Case Spezifikation


Anwendungsfälle sind in Gruppen organisiert. Die erste Gruppe (in der Ausarbeitung Abschnitt 6.1) stellt die Funktionalitäten / die Eigenschaften vor, die erforderlich sind, die grundlegende Programmlogik zu fassen, sie enthält:
1. Abbildung von Java Prozessen und Kontrollfäden auf das UPPAAL Modell,
2. Abbildung von funktionalen Java-Programelementen auf zeitgesteuerten Automaten (Zustände/Übergänge) in UPPAAL,
3. Abbildung von objektorientierten Java-Programelementen auf zeitgesteuerten Automaten (Zustände/Übergänge) in UPPAAL,
4. Nutzung von extrahierten PV-Toolvariablen,
5. Bibliotheken für Pseudocode, sowie Extraktion von UPPAAL Pseudocode und die Abbildung auf UPPAAL.

Die nächste Gruppe wie sie im Abschnitt 6.2 der Ausarbeitung vorgestellt wird, führt Funktionalität für die Modellierung von Netzwerkkommunikationsmustern (mit Echtzeitaspekten) ein.

- Entfernte Methodenaufrufe mit Java RMI,
- Kommunikation mit Nachrichtenversand,

Abschließend stellt die dritte Gruppe (vgl. Abschnitt 6.3) Funktionalitäten vor, zur Überprüfung des Modells gegen Eigenschaften/Anforderungen verwendet wird:

- Erreichbarkeitsanalyse und Abwesenheit von Verklemmungen,
- Vorbedingungen, Nachbedingungen und Zusicherungen,
- Invarianten,
- Komplexe zeitliche Logikausdrücke.

Die Anwendungsfälle (abgesehen von denen in Abschnitt 6.3) werden in der gleichen strukturierten Methode spezifiziert. Im ersten Anwendungsfall (vgl. Abschnitt 6.1.1) wird die Struktur detailliert beschrieben.

- Übersicht und Gründe,
- Analyse,
- Anwendungsfall,
- PV Input und Output,
- PV-Tool
- Fehler und Warnungen.
Beispiele einer Use-Case Spezifikation

Jeder Anwendungsfall spezifiziert eine neue Funktionalität, die das PV-Tool bereitstellen muss. Die Anwendungsfälle weisen dabei die gleiche Struktur auf. Zuerst wird eine Zusammenfassung und eine Begründung für die Notwendigkeit des Anwendungsfalls diskutiert. Danach beschreibt der Abschnitt Analyse die Hauptprobleme die adressiert, Vereinfachungen die genommen und das ausgewählte Verfahren, das eingesetzt wird. Der Abschnitt Test Case gibt ein konkretes Beispiel einer Anwendung. Dieses Beispiel wird während der Spezifizierung des Anwendungsfalls (im folgenden Abschnitt) verwendet. Input und Output spezifiziert die Eingaben für des PV-Tool (Quellencode und Annotationen), sowie die entsprechende Ausgabe (UPPAAL Modell und Eigenschaften). Lassen Sie uns einige Beispiele näher betrachten:

Abbildung 6-89: RMI Aufruf mit dem Datenaustausch: Input (Dienstnehmer)

Auf der Dienstgeber-Seite haben wir eine entfernte Methode:

Abbildung 6-90: RMI Aufruf mit dem Datenaustausch: Input (Dienstgeber)

Als Ausgabe wird ein Timed-Automata erzeugt (außer Deklarationen, die wir hier nicht zeigen). Im Beispiel werden für den Dienstnehmer die folgenden neuen Transitionen erzeugt:

Abbildung 6-91: RMI Aufruf mit dem Datenaustausch: Output (Dienstnehmer).

Dieser Anwendungsfall spezifiziert neue Annotationen, wie zum Beispiel @rmiLookup. Die Tabelle 6-24 listet Eigenschaften zur Verwendung und mögliche Attribute genauer auf.
Im Anschluss daran spezifiziert der Abschnitt *PV Tool* die notwendigen Designerweiterungen innerhalb des PV-Tools, um den spezifizierten Anwendungsfall zu realisieren. In einem Beispiel (Nicht-RMI) wird das Generator Modell erweitert:

**Abbildung 6-16: Generator meta-model**

Schließlich spezifiziert Abschnitt *Errors and Warnings* das Verhalten des PV-Tools unter anormalen Bedingungen und im Fehlerfall.
Komplette Validierungsanwendung

Um die Nützlichkeit der Anwendungsfälle zu demonstrieren, stellen wir hier ein Beispiel der Anwendungen dar, das mit dem PV-Tool modelliert wird und im Anschluss mit UPPAAL analysiert werden kann.

Dazu verwenden wir eine Variante einer Beispielanwendung, die mit UPPAAL ausgeliefert wird: Das Train-Gate Beispiel beschreibt die Nutzung einer Eisenbahnweiche durch mehrere Zug-Objekte.

Innerhalb dieses Anwendungsfallbeispiels verwenden wir verschiedene Annotationen und die Funktionalitäten, die vom PV-Tool bereitgestellt werden:

1. Klassen, Objekte, Methoden, Attribute,
2. Schleifen und Bedingungen,
3. Nachrichten-basierte Kommunikation,
4. Bibliotheken,
5. Eigenschaften/Anforderungen.

Wenn möglich, verwenden wir Standardwerte für Annotationen um zu zeigen, dass nur wenige Annotationen im Programmcode in der Tat erforderlich sind.

Für unser Testverfahren konstruieren wir das Beispiel zuerst in Java. Um dies zu tun, folgen wir der Java „Philosophie“ beim Schreiben des Codes. Dann erstellen wir das resultierende PV-Tool Modell eigenhändig (das Werkzeug war noch nicht völlig implementiert), aber wir folgen dabei den Anwendungsfallbedingungen.

Dies ist genau die typische Anwendung des Tools, wenn kein vollständiges Design existiert, aber Teile innerhalb des Quellprogramms entwickelt und analysiert werden sollen.

Implementierungsbericht

Die Entwicklung einer durchgängigen Spezifikation und notwendiger Bedingungen für ein PV-Tool war Inhalt dieser Diplomarbeit. Um die Forschung und die Spezifikation des PV-Tools zu ermöglichen wurde eine „Proof-of-Concept“ Implementierung umgesetzt.


Das PV-Tool bietet als grafische Oberfläche ein Plug-In für IBM Eclipse. Dies ermöglicht es dem Entwickler das PV-Tool einfach zu konfigurieren, ausführen und Fehlerliste einsehen zu können. Das folgende Bild zeigt ein Konfigurationsfenster:
Weitere Arbeit

Innerhalb der Diplomarbeit stellten wir Anwendungsfälle um, die höchste Priorität haben. Jedoch sehen wir, dass weitere Arbeit erledigt werden kann, insbesondere in den folgenden Bereichen ist dies denkbar:

1. Tiefere Analyse der objektorientierten Aspekte,
2. Weitere Kommunikationsmodi (z.B. unzuverlässiger Datenaustausch, Rundruf),
3. Automatische Modellieren von Standard-Netzwerk-Stacks (wie TCP, UDP, IP), so dass der Entwickler weniger Annotationen selbst schreiben muss,
4. Modellierung von Zeiteinschränkungen (wie Timeouts),
5. Synchronisierte Blocke/Methoden (innerhalb von Prozessen),
6. Thread-Synchronisierungsmechanismen (wie z.B. wait/notify),

Weitere Erweiterungen können eingeführt werden, nachdem das Feedback von Entwicklern (Benutzern) entsprechend ausgewertet ist. Außerdem kommen dem PV-Tool die Änderungen in UPPAAL und Erweiterungen dort zu Gute.
Validierung und Zusammenfassung


Dies hat sich als sehr sinnvoll für das PV-Tool erwiesen:


2. Wir identifizierten außerdem unwichtigere Fehler, Unstimmigkeiten und Mehrdeutigkeiten innerhalb der Anwendungsfälle und konnten diese entsprechend aktualisierten.

Hinsichtlich des Beispiels selbst, zeigten wir, dass das Verfahren erfolgreich arbeitet und Fehler in verteilter Anwendung effizient finden kann. Durch die Überprüfung der Eigenschaften wurden die gefunden, die nicht erfüllt waren. Außerdem war das PV-Tool in der Lage auch „dumme“ Fehler im Programmcode zu identifizieren (Beispiele dafür sind falsche bedingte Anweisung oder die fehlerhafte Verwendungen von Array-Objekten, die in eine Verklemmung resultierten). Diese Fehler wurden durch die Modellsimulation gefunden, die zufällig alle möglichen Übergänge des Zustandsautomaten untersucht, bzw. durch eine manuelle Simulation, die durch die Auswahl von Übergängen durch Entwickler erfolgte. Wir arbeiteten mit der Fehlersuche bis alle Eigenschaften erfüllt waren. So kam heraus, dass die grafische Schnittstelle und die benutzergesteuerte Simulation der zeitgesteuerten Automaten zusammen mit der zufallsgesteuerter Simulation eine sehr effiziente Methode für das Auffinden von Fehlern darstellt.
# Table of Contents

**Abstract** .................................................................................................................................................. 5

**Kurzfassung (Deutsch)** ......................................................................................................................... 7

Einführung .................................................................................................................................................. 7

Analyse ..................................................................................................................................................... 8

Aktuelle Technologien ................................................................................................................................. 9

Konzept ................................................................................................................................................... 10

Softwarearchitektur .................................................................................................................................. 11

Use-Case Spezifikation .............................................................................................................................. 14

Beispiele einer Use-Case Spezifikation .................................................................................................... 16

Komplettes Validierungsanwendung ........................................................................................................ 18

Implementierungsbericht ........................................................................................................................... 18

Weitere Arbeit ......................................................................................................................................... 19

Validierung und Zusammenfassung ......................................................................................................... 20

**Table of Contents** ............................................................................................................................... 21

1 **Introduction** ....................................................................................................................................... 23

1.1 Application Areas ................................................................................................................................. 24

1.2 Document Structure ............................................................................................................................. 24

2 **Analysis** .......................................................................................................................................... 25

2.1 Model Checking based on Design ....................................................................................................... 25

2.2 Model Analysis based on Source Code ............................................................................................... 26

2.3 Properties .......................................................................................................................................... 27

2.4 Model ................................................................................................................................................ 28

2.5 Summary of Analysis ........................................................................................................................... 28

3 **State of Art** ...................................................................................................................................... 30

3.1 Model-Checking Tools ......................................................................................................................... 30

3.2 Tools Extracting Model from Code ..................................................................................................... 31

3.3 Tools for Model-Driven Development .................................................................................................. 33

4 **Requirements and Overall Concept** ............................................................................................... 35
5 Architecture Specification ................................................................. 37
6 Use-Case Driven Specification ......................................................... 40
   6.1 Program Structure ....................................................................... 41
   6.2 Inter/Intra Process Communication .............................................. 114
   6.3 Model Verification against Properties ........................................... 141
7 Complete Use Case Example ............................................................. 145
   7.1 Functional Overview of the Application ........................................ 146
   7.2 PV Tool Input – Annotated Java Code ........................................... 147
8 Report on the Implementation ............................................................ 155
9 Further Work .................................................................................. 157
10 Verification, Validation and Conclusion ........................................... 158
11 References ..................................................................................... 159
   11.1 Documents .................................................................................. 159
   11.2 Acronyms .................................................................................... 160
12 Annex A: Specification of Annotations ............................................. 161
   12.1 General Annotation Syntax ......................................................... 161
   12.2 PV Tool Annotations ................................................................. 162
1 Introduction

In mission-critical systems, it is important to keep the software errors down to minimum. For coding/development errors there are various efficient and frequently used techniques like compiler checks, debugging, assertions, coding guidelines checks, unit testing, which enable to detect errors soon after they occur.

However, for finding errors in design, there are fewer techniques available. There main ones are integration testing, simulation or design verification. The main problem with integration testing is that it occurs at the very late stage of development cycle, where removing of errors is very expensive. Simulation can be executed only a bit earlier, but it requires to write much simulation wrappers (like for unit testing), which causes major difficulties for complex distributed applications. Moreover, the coverage of both methods is limited due to exponential number of system states and difficulty of simulating/having various ranges of states and conditions.

For design verification, there are also peer document reviews (or document inspections) or more structured design verification analysis like Software HAZOP, FMEA. However, most of the added value from them is only independent review by people having different perspective and experiences.

However, these methods used in the early stages of development very ad-hoc, and insufficient due to low error detection coverage, “as they are based mainly on natural language specifications and informal or semi-formal diagrammatic notations” [R3].

Formal modeling is targets to improve the system analysis phase. It helps to achieve a better understanding of requirements and to reduce risks. Formal methods can be applied to verify the design at early stages of the lifecycle. They are very efficient in finding protocol errors and concurrency-related errors in non-deterministic and highly distributed systems. However, the problem is that in most projects, even these mission-critical ones, the design is not complete or precise enough, or it is not reflecting the source code (because of synchronization costs). Moreover, usage of model checking of design requires formalization of the design specification and of requirements – this requires experts in formal methods.

An alternative approach is to use formal methods directly on the source code rather than on design: the model is extracted automatically or semi-automatically (with support of developers) from the source code and then verified against properties. The semi-automatic approach is the one selected by the tool that is specified within this document.

In the recent years seeing very low adoption of formal methods in software development (contrary to hardware development), the goal is no longer seen as proving that a system is bug-free. Rather, the goal is to provide a cost-effective solution to find early design errors.
As a result, we propose within this document a new approach: a semi-automatic model extraction, addressing the mentioned problems. It is the developer who selects which parts of code are relevant for model checking, and selects how to abstract them. The developer needs simply to annotate Java statements with code comments. Within this diploma thesis, we specify the new Protocol Verifier Tool (hereafter called PV tool), which demonstrates the validity of our approach.

1.1 Application Areas

Any new tool or technology can be used only if it is justified - it brings sufficiently added value (in our case it is finding efficiently undiscovered new bugs), and has sufficiently low integration costs (in our case it is learning curve, time spent to do annotations and to analyze error traces). The application areas where the tool is useful include “safety critical applications (e.g. railway switches), security critical applications (e.g. access control, electronic banking), cost critical applications (which, for example, run on a large number of non-administrated devices, such as phone cards), and legally critical applications (e.g. falling under digital signature laws).” [R18], which can be summarized as “mission-critical” systems.

1.2 Document Structure

Chapter 1 (Introduction) gives an overall introduction and states the problems and issues to solve.
Chapter 2 (Analysis) is an in-depth analysis of the problem
Chapter 3 (State of Art) gives an overview of related state-of-art
Chapter 4 (Requirements and Overall Concept) gives an introduction of the PV tool
Chapter 5 (Architecture Specification) specifies the architecture of the proposed tool
Chapter 6 (Use-Case Driven Specification) specifies use cases specifying the functionalities of PV tool
Chapter 7 (Complete Use Case Example) gives a complete example demonstrating usage of the PV tool
Chapter 8 (Report on the Implementation) summarizes the work done in PV tool implementation
Chapter 9 (Further Work) points the work that needs to be done in future
Chapter 10 (Verification, Validation and Conclusion) gives a conclusion and validates the results against the initial goals
Chapter 11 (References) gives references to literature and to source code and examples
Chapter 12 (Annex A: Specification of Annotations) is the annex specifying all PV tool annotations
2 Analysis

Within this chapter we give an overview of model checking done based on design and on source code. We also give overview and classification of notations/languages for models and properties.

In model checking, there are two main entities: model, which is a formal description of the system behavior, built based on design or based on code and secondly properties, which represent formalized requirements on the system.

2.1 Model Checking based on Design

“Model checking is a method to algorithmically verify formal systems. This is achieved by verifying if the model, often deriving from a hardware or software design, satisfies a formal specification.” [R11].

The typical application of formal methods is to verify the formal model against formal properties. In more details, the flow is as follows (see Figure 2-1):

1. The textual requirements are formalized into so-called properties, which are formal expressions expressed using a logic,
2. The design descriptions (in form of diagrams, textual descriptions, examples) are formalized into a formal model, specifying the design in an abstract, formal description language,
3. Once the properties and model are available, the model is verified against the properties,
4. As a result, for each property:
   a. There is a positive statement that the property is verified,
   b. Or there is a statement that the property is not verified and the way(s) to reach the error state (state when the property is violated) is returned.

In case a property is violated, it usually means that there is a design error. But it can also mean that the property itself is incorrect.

The presented approach can be applied to verify the design at very early stage of the lifecycle. The source code does not even have to exist. Moreover, it is very efficient in finding high-level errors, due to high-level of abstraction of the design and of the model. It is especially useful for detecting communication errors.
However, the problem is that in most projects, even these mission-critical ones, the design is not complete/precise enough to be able to do a model that is complete enough. Moreover, the design within time tends to get obsolete within time, when the code is developed. Moreover, this approach requires experts in formal methods.

As a result, formal model checking failed to win a broad popularity in industry. Such approach is much used in avionics and aerospace, an example of usage in other industries is the formal verification of safety protocol PROFIsafe [R19].

![Figure 2-1: Design-based model checking](image)

**2.2 Model Analysis based on Source Code**

An alternative approach to address the issues with model checking based on design is to extract the formal model automatically (or semi-automatically) from the source code (with a model extraction tool) and then verify this formal model against properties.

In the automatic model extraction, “bridging the semantic gap between a non-finite-state software system expressed as source code” and formal model “requires the application of sophisticated program analysis, abstraction, and transformation techniques.” [R13]. Alternatively, as proposed by this diploma thesis, the model extraction can be guided by developer, enabling to reach higher level of abstraction and reduce the model to only relevant elements.

In more details, the flow is similar to the design-based one (see Figure 2-1). The steps that are different than in the previous approach are bold:

1. The textual requirements are formalized into so-called properties, which are formal expressions expressed using a logic,

2. The code is implemented based on complete or partial design and based on requirements,
3. The code are formalized into a formal model using a model extraction tool,
4. Once the properties and model are available, the model is verified against the properties,
5. As a result, for each property:
   c. There is a positive statement that the property is verified,
   d. Or there is a statement that the property is not verified and the way(s) to reach the error state (state when the property is violated) is returned.

“The type of abstraction that is appropriate for a given application depends both on the type of logical properties that we are interested in proving and on the resource limits of the verification system.” [R3]. Therefore, “for the best choice of an abstraction method in the construction of a verification model, we unavoidably have to rely on human judgment. Which parts of the system should we look at? What properties should apply? These types of decisions would be hard to automate. But even though there will unavoidably be a human element in the setup of a verification process, once the basic decisions about abstractions are made and recorded, it should in principle be possible to mechanize the remainder of the verification process.” [R3]. This is exactly the approach that the PV tool will follow – developer supported model extraction. Thanks to this approach, the developer can reach a simple, clean output model, which is understandable and technically verifiable.

### 2.3 Properties

As already mentioned, properties are formalized system requirements. A critical and step is to “decide what precisely the critical correctness properties of the application are” [R3], i.e. which really need to be analyzed by model checking.

The properties are expressed as temporal logic formulas. Two types of temporal logic are used:

1. Linear Time Logics – (e.g. LTL),
2. Branching Time Logics – (e.g. CTL),

Computational tree logic (CTL) is a type of Branching Time Logic. It is also referred as Computation tree logic. It is often used to express properties of a system in the context of formal verification or model checking.

The logical operators are the usual ones (NOT, AND etc). Along with these operators CTL formulas can also make use of the boolean constants true and false. The temporal operators are the following:

1. State operators:
   a. $\forall \varphi$ - All: $\varphi$ has to hold on all paths starting from the current state.
   b. $\exists \varphi$ - Exists: there exists at least one path starting from the current state where $\varphi$ holds.

2. Path operators:
a. \( N \varphi \) - Next: \( \varphi \) has to hold at the next state (this operator is sometimes noted \( X \) instead of \( N \)).

b. \( G \varphi \) - Globally: \( \varphi \) has to hold on the entire subsequent path.

c. \( F \varphi \) - Finally: \( \varphi \) eventually has to hold (somewhere on the subsequent path).

d. \( \varphi U \psi \) - Until: \( \varphi \) has to hold until at some position \( \psi \) holds. This implies that \( \psi \) will be verified in the future.

e. \( \varphi W \psi \) - Weak until: \( \varphi \) has to hold until \( \psi \) holds. The difference with \( U \) is that there is no guarantee that \( \psi \) will ever be verified. The \( W \) operator is sometimes called "unless". (based on CTL definition from [R11]).

According to [R20], “while branching time logic is suitable for reasoning about nondeterministic programs, linear time logic is preferable for reasoning about concurrent programs”.

The following types of properties have been identified (list based on [R3] and other sources):

1. Invariants - conditions that remain unchanged;
2. Assertions - conditions that apply at a single place;
3. Preconditions and postconditions - conditions that apply before and after something occurs in a system. These can be viewed as assertion pairs;
4. Constraints - statements that apply across changes that occur in the system.

### 2.4 Model

When creating the model based on design, or when doing semi-automatic model extraction from source code, the developer/modeler should try to define the smallest sufficient portion of the system that to cause that the properties of interest are satisfied.

There are several tools on the market, but from the mathematical/logical perspective, they use one of the following formal models:

1. timed automata or
2. finite automata or
3. hybrid automata.

Automata can have a textual notation (like PROMELA) or graphical notation (like UPPAAL).

### 2.5 Summary of Analysis

The main issue with Model Checking based on Design is that usually the design is not complete or precise enough, or it is not reflecting the source code (because of synchronization costs). Moreover, usage of model checking of design requires formalization of the design specification and of requirements – this requires experts in formal methods.
The main issue with automatic model extraction from code is that the extraction tool does not know which information is relevant and to which degree the abstraction should be done. As a result, the tools doing automatic model extraction suffer from so-called “state-explosion” and are not scalable.

As a result, we propose within this document a new approach: a semi-automatic model extraction, addressing the mentioned problems. It is the developer who selects which parts of code are relevant for model checking, and selects how to abstract them. The developer needs simply to annotate Java statements with code comments.

We have presented of classification of languages for defining formal methods and languages for defining properties. However, from the mathematical/logical perspective there is no clear leader: the choice of UPPAAL within this project is due to overall considerations (including user-friendliness), and not just the mathematical background.
3 State of Art

This chapter gives an overview of existing tools/solutions within the area of PV tool, which are formal and semiformal design, model checking.

First, we present model checking tools (section 3.1), then tools extracting model from code (section 3.2), and then tools for model-driven development (section 3.3).

3.1 Model-Checking Tools

Model-checking tools verify the formal model against the properties. For defining the model and properties, the model checking tools have their own formal language. Moreover, the tools have a very heavy mathematic background.

There are many model-checking tools (more than 50). There are commercial/free/open-source tools, with various goals and functionalities. In this chapter, we present only selected ones.

3.1.1 SPIN

SPIN is a “popular open-source software tool, used by thousands of people worldwide, that can be used for the formal verification of distributed software systems. The tool was developed at Bell Labs in the original UNIX group of the Computing Sciences Research Center, starting in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field. In April 2002 the tool was awarded the prestigious System Software Award for 2001 by the ACM.” [R10].

SPIN model specification language, called PROMELA, is a non-object-oriented textual language. The tool itself is a set of command-line utility programs and to check the model, the developer needs to have a script/makefile.

3.1.2 UPPAAL

UPPAAL is “an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types”[R21].

UPPAAL model specification is a graphical language (timed automata). The main advantage of it is easy bug finding through simulation. Moreover, parts of the model can also be specified in pseudocode. This enables to keep the model compact, model graphically only “important” aspects of the model, and express as pseudocode the parts of model that are just needed for its functioning (like fifo queue utility class).
UPPAAL is very actively developed, with new releases and features coming every few months. It has a powerful graphical front-end, but can also be used from command line.

UPPAAL input language is serialized to XML for model and to TXT for properties. Thanks to this, it is very straightforward to generate from a tool a correct UPPAAL model: one just needs to generate a valid XML.

## 3.2 Tools Extracting Model from Code

Model checking tools have had a very low adoption by industry. Apart from aerospace industry, they are not widely used. Even if usage of formal methods is recommended by safety standard IEC 61508 [R14], they are rarely used in SIL 2 and SIL 3 systems (in short, SIL2 and SIL3 systems are safety-relevant systems with protect control systems that could otherwise cause severe harm including death to several persons).

Moreover, in most of embedded/safety related/dependable projects, there is no correct development process followed, and there is no complete/correct software design. Without having a design, it is impossible to formalize the design into a formal model. Moreover, doing the design post-facto (documenting a source code) is expensive and in most case will have big differences (code done without clear design tends to be spaghetti-like). Therefore, formal verification of the design which is different that the real source code has little sense.

As a result, there are some tools on the market (including the PV tool specified in this document), which assume that the correct development process is indeed not followed and the design is not available. Instead, the model is automatically (or semi-automatically) extracted from the existing source code by a tool and then simulated or verified against properties.

There are several tools for C, which is a standard language for embedded systems. Within this chapter, we concentrate only on tools extracting from Java code, as only they are relevant for PV tool.

### 3.2.1 JavaPathFinder

JavaPathFinder is “is a system to verify executable Java bytecode programs. In its basic form, it is a Java Virtual Machine (JVM) that is used as an explicit state software model checker, systematically exploring all potential execution paths of a program to find violations of properties like deadlocks or unhandled exceptions.” [R12].

JavaPathFinder automatically translates Java bytecode into PROMELA (SPIN model specification language). To define the properties, the developer writes special test classes. “JPF can search for deadlocks and unhandled exceptions (e.g. NullPointerExceptions and ASSERTionErrors), but the user can provide own property classes, or write listener-extensions to implement other property checks (like race conditions)” [R12].

Due to fully automatic extraction of the model, without guidance what is important and what is not, the tool suffers from “state explosion” and is not scalable. Programs up to 10kloc can be extracted and they generate very complex models.

The following figure gives a data flow overview of JavaPathFinder tool. The compiled Java program, extended with properties written in Java is processed by the tool. As a result, the verification report shows the violations of properties.
3.2.2 Bandera

Bandera is another leading tool for model extraction from Java code. It has “an open architecture in which a variety of analysis and transformation components may be incorporated.” [R13].

Bandera supports several model checking tools (not just UPPAAL, like JPF). Moreover, it uses Java source code (rather than Java bytecode) as input:
3.3 Tools for Model-Driven Development

A big alternative to model extraction from source code is to follow a model-driven development (MDD). In model driven development, the software specification is a model that tightly reflects the source code. Rather than creating ambiguous specification document, the developer/designer creates a semi-formal model - a semi-formal specification. This model is then two-way synchronizable by the tool (model → code and code → model).

The de-facto standard for semi-formal modeling is UML, extendable with various profiles (like DoDAF). UPPAAL has also a textual specification language, called OCL.

Note that UML itself can be used in very different ways. It is very flexible, and one can use it for describing very precise semi-formal models, and another can just describe ambiguous examples for his design documentation.

Note also that UML semantics is NOT formal and it depends on the tool used.
3.3.1 Rhapsody and Rational Rose RT

IBM Rational Software Architect [R17] and I-Logix Rhapsody [R16] are the two leaders in model-driven development. They provide two-way synchronization between model and code; they offer integration into Java IDE (Eclipse). The synchronization is advanced, and does not cover just class diagrams (as for majority of simple UML editors) but also cover state diagrams, activity diagrams and others. Finally, these tools are integrated IDEs, and enable to design, code and build/compile the programs.

Note that Rhapsody and Software Architect do not offer the model-checking.

3.3.2 KeY

KeY tool is plug-in for another integrated MDD tool, Together Control Center. [R18]. It extends the semantics of the model with JML and OCL constraints.

Rather than converting the model and code into a formal model (like PROMELA model), the KeY tool converts it into a sequence of mathematical formulas (DL, i.e. Dynamic Logic). Then, another theorem prover tool proves it, verifying by itself the properties.

![KeY tool flow](R18)

The KeY tool suffers from lack of abstraction. Moreover, it has limited feature set: for example it does not support concurrency (threads).
4 Requirements and Overall Concept

As a result of the analysis, and state of the art, we come up to the following requirements for the Protocol Verified tool (called hereafter PV tool), followed by the overall concept. The requirements are fulfilled by the following chapters (Architecture, Use-Case Driven Specification). Due to the constrained size of the document and diploma thesis context, we do not demonstrate that the requirements are fulfilled.

1. The tool shall support Java 2 SE as input language with annotations provided by developer;
2. The tool shall support programs where not the whole source code is available (some parts can be in system libraries or external libraries);
3. The annotation notations shall be easy for developers, and shall use Java standards for the syntax;
4. The tool shall provide a intuitive default behavior in case annotations or parts of annotations are not provided;
5. The tool shall check annotations provided by the developer and verify them for correctness and then notify the developer in case of error;
6. The tool shall generate UPPAAL model/properties as output;
7. It shall be easy for developer to interpret/understand the output model and the error trace;
8. The entities within the model shall have traceability back to source code (so that the developer can understand where they come from);
9. The tool shall identify design/implementation errors (and not prove the correctness);
10. The tool must be fast and scalable up to big Java projects.

The main functionalities differentiating our approach from the state of the art are as follows. First, we provide a tailored solution, for model checking of Java programs using UPPAAL. Thanks to this, we are able to provide a simple and efficient solution. Moreover, we are giving the possibility to analyze/simulate the model with a powerful graphical UPPAAL model simulator. We also do not define new language for specifying properties: we use UPPAAL property syntax.

Secondly, in the correct abstraction of the source code, the PV tool depends much on developers – the developers are supposed to annotate Java code, providing the selection of elements that relevant and abstraction hints. This makes the PV tool much simpler, gives more flexibility to developer, enables to model only relevant parts, enables to avoid state explosion and enables to generate at the end a simple, understandable output (a model), which can be analyzed by developer through graphical simulation.
Moreover, even if selection and abstraction decisions are done by developer, all extraction tasks that can be automated are to be provided by PV tool, enabling to save much time and avoid unnecessary human errors.

Finally, the approach focuses mostly on high-level modeling of the application and provides high level of abstraction. The core concern is correct modeling of communication in distributed systems.

Further advantage of our approach is first that the annotations can be written by developer. They are simple and intuitive for Java developer - we respect standards (Javadoc and JML syntax) [R6]. This also enables smooth integration with Java IDEs and Javadoc generator. Due to the fact that the annotations are within the source code and because they are both done by developers, it is easy to keep them synchronized.

The syntax for property annotations is simple and the developers do not have to study formal methods literature.

We use as backend a UPPAAL, which has advanced GUI interface, and enables to simulate graphically the timed automata. Moreover, UPPAAL is very actively developed and there are constantly new features.
5 Architecture Specification

This chapter describes the overall architecture of the Protocol Verifier Tool (hereafter called PV tool). It gives first an overall functional description of the toolchain and later describes the some most important aspects of the tool architecture, some its logics and the intermediary program abstraction models. Based on this architecture specification, the next chapter (chapter 6) gives more detailed specifications organized by use cases (i.e. major functionalities).

The input for the tool is annotated Java source code with the properties. The tool extracts the UPPAAL model and properties out of the input. Then, the model is verified against the properties by UPPAAL. As a result, there are verification results (yes/no for each property) and an error trace in case errors were found (see Figure 5-1):

Figure 5-1: Summary of data flow, from Java code to results

In more details, the model extraction is as follows (see Figure 5-2). First, an external code processing library is used (Recoder), which parses the source code (the text) and builds low-level model representing this source code. Therefore, thanks to Recoder, we access to Java source code through an API.

Once we have Java data objects representing the source code, we parse it: the Parser subsystem creates an intermediary abstraction model, which models the code as sequence of statements.
Based on the intermediary model created by Parser, Generator subsystem creates the Abstracted Application Model, which is yet at higher level abstraction. Within this model, we do not have just a list of statements in a sequence. Rather, the generator model defines in a hierarchical way the code sequences.

The Abstracted application model is then transformed to Final PV model. The final PV model is a Java representation of UPPAAL XML document structure.

At the end, the final PV model is simply serialized using an external library (XBean).

**Figure 5-2: Summary of model extraction step (from Java to UPPAAL model)**

The main control subsystems and metamodel subsystems within PV tool are as follows (Figure 5-3):

1. Parser: responsible for generating intermediary model (i.e. Parser model),
2. Parser model: defining the class hierarchy to model code as sequence of actions (statements),
3. Generator: responsible for generating abstract application model (i.e. Generator model),
4. Generator model: generating itself the final PV model, and defining the class hierarchy to model the program execution in a hierarchical, abstract way,
5. Final PV model: direct representation of UPPAL files before serialization on XML files.

Figure 5-3: Most important elements in the PV tool
6 Use-Case Driven Specification

The use-case driven specification is the core of this thesis. Based on the initial overall tool architecture, as specified in chapter 5, various functionalities are introduced to fulfill the requirements and overall concept specified in chapter 4.

The use cases are ordered by their importance and impact on the tool architecture, and they are supposed to be implemented in this order. In other words, the approach is architecture-centric and risk-driven. Some use cases are more or less complex, bigger and smaller – the split into use cases is done to provide a self-contained new functionality/property rather than equally-sized work packages. The order of implementation of use cases is therefore mostly imposed, as one use-case assumes that features provided by another one are already available.

The use cases are organized into groups. The first group (section 6.1) introduces the functionalities/properties that are needed to have the basic program structure, which includes:

1. Mapping of Java processes and threads to UPPAAL model,
2. Mapping of functional Java program elements to UPPAAL timed automata locations/transitions,
3. Mapping of object-oriented Java program elements to UPPAAL timed automata locations/ transitions,
4. Usage of parseable PV tool variables,
5. Pseudocode libraries and extraction of UPPAAL pseudocode and mapping to UPPAAL

The next group (section 6.2) introduces the functionalities to specifically support and model various network communication patterns (including where realtime aspects):

1. Remote method invocation,
2. Communication using message passing,

Finally, the group (section 6.3) introduces the functionalities enabling verification of the model against properties:

1. Reachability and deadlocks,
2. Preconditions, postconditions and assertions,
3. Invariants,
4. Complex temporal logic expressions.
The use cases (apart from those specified in 6.3) are specified in the same structured way, described in more details in the first use case (section 6.1.1), which can be summarized as follows:

1. Overview and rationale (overview of the use case and the rationale behind it),
2. Analysis (analysis of main issues, challenges, aspects related to the use case),
3. Test case (description of one or more, typical example(s), which is used to help explaining the specification),
4. Specification of inputs and outputs,
5. Design extensions of the PV tool due to the use case,
6. New types of errors and warnings introduced by the use case.

6.1 Program Structure

This group of use cases introduces the functionalities/properties that are needed to have the basic program structure, which includes:

1. Mapping of Java processes and threads to UPPAAL model,
2. Mapping of functional Java program elements to UPPAAL timed automata locations/transitions,
3. Mapping of object-oriented Java program elements to UPPAAL timed automata locations/transitions,
4. Usage of parseable PV tool variables,
5. Pseudocode libraries and extraction of UPPAAL pseudocode and mapping to UPPAAL

6.1.1 Processes and Threads

Section “Overview and Rationale” gives an introduction of the use case, presents its reasons of existence and goals to be reached.

At the beginning, we need to define what a distributed application is. Taking into account that we model the dynamic behavior of a distributed application (as opposed to the static one), firstly we need to differentiate between processes and threads – this is because only within the threads of one process we can have shared memory and shared memory communication. Between processes, there can be only inter-process communication (like RMI, Sockets). Secondly, we need to differentiate between processes, threads and on the other side process instances and respectively thread instances. It happens that one process exists in several instances in one distributed application (like several instances of a web client browser). In the same way, within one process, one thread can be in several instances (like several Servant threads handling RMI calls).

To keep the model simple, we don’t introduce separate classes to model the instances (process instances and thread instances). We only keep one attribute (quantityOfInstances) which stores the quantity of instances of a particular process or thread.
More precisely, a distributed application is made of processes, existing in several instances. Each process instance is a parameterized instance of process. Moreover, a process has within its scope several threads, each existing in several instances.

The following object diagram gives an example of a particular system.

In the above example, process Client exists in 10 instances, meaning that there are 10 clients at runtime. Each client process has one GUI thread, which gives in total 10 GUI threads.

In UPPAAL, the basic entities building up the application is a template, corresponding to a thread.
A template is a timed automaton (where a timed automaton is a state machine extended with time modeling). The system is composed of instances (instantiations) of templates.

There is no concept of processes within UPPAAL. We will model the processes by the naming conventions of templates, instances, variables (with prefixes). Moreover, only thread instances of the same process instance will communicate over shared memory.

Some examples covered by the described model are as follows:

1. One or many instances of the same process: several identical clients (processes) communicating with one server;
2. Instances of different threads within one process: several servant threads handling remote method calls.

6.1.1.1 Analysis

Section Analysis describes main issues to be addressed, overall simplifications and approach that is taken.

The first issue is that UPPAAL does not have a separate concept for processes and process instances - there are only threads (i.e. templates) and thread instances (i.e. systems). However, it is quite straightforward to model processes and process instances with just naming conventions and constraints on using shared data.

The next point is that one thread can exist in several instances within each process instance. Each thread instance is unique and must be differentiated and easily identified. Moreover, each thread instance has its custom parameters.

Moreover, the information about the quantity of processes to create is not written within Java code and a priori is unknown (e.g. number of client web browsers). It is to the developer to decide how many instances are to be created to test the model.

Another problem is dynamically created threads. In Java, one can create a thread dynamically, with an explicit function call (e.g. `Thread.start(new MyRunnable())`). A thread can also be created by Java middleware (Servant threads are created on demand, depending on the number of concurrent remote client calls).
It is proposed to map all threads to UPPAAL templates. Thread instances are mapped to template instances. The processes and process instances are mapped to separate memory spaces through naming convention and restricted usage of shared variables.

The proposed model makes the following main simplifications:

1. The number of process and thread instances is fixed and specified for each process by an annotation;
2. The thread instances and process instances are all created statically;

The processes are identified by means of `main()` methods. The user selects which `main()` methods belong to the application. Concerning threads, there are three types of threads considered:

1. Thread of `main()` method (note that `main()` represents both a process and a thread;  
2. Thread initiated by `run()` method of `Runnable` class;
3. Thread for handling remote RMI method (a servant thread), initiated by each remote call by RMI middleware.

The modeling of Remote methods (the third case) is the most complex one. A method belongs to an interface, an interface is realized by a class, and the class is instantiated by several objects. We simplify this, assuming that there is just one class implementing the Remote interface. However, we model that there are several objects for one interface. Each remote method for each object forms several thread instances.

Finally, synchronization is not covered in this use case: if methods of the interface are synchronized, then only one can run at a time. This will be covered in further use cases.

### 6.1.1.2 Test Case

Section Test Case gives a concrete example of an application. This one example is used throughout the specification of the use case (the next section) to give an example for every modified aspect of the PV tool.

The example is as follows: there is one Bank server and two cash withdrawal devices (i.e. ATMs). The ATMs communicate with the Bank server over RMI. Each ATM has one main thread and one communication thread in two instances. The bank has one main thread, and the threads resulting from two bank accounts (objects): each account has two methods, and each method makes two thread instances.

![Figure 6-4: Test case processes and threads](image-url)
6.1.1.3 PV Input

The input of the PV tool can be classified as follows:

1. Java source code – main source of the information;
2. External/command line parameters – for some initial configuration, like selection of processes.
3. Custom Javadoc annotations – for facilitating the model extraction and for providing information not present within source code;

External Parameters

The external parameters are parameters as follows:

1. Command line parameters,
2. Parameters from the GUI front-end of PV tool.

To specify which processes belong to the distributed application, the fully qualified paths (package path + class name + method name) of main() methods are provided. This is the main configuration parameter, which is provided as a command line configuration parameter to the tool. This is to enable to select the processes that are to be considered from the whole set of processes that is available.

The second parameter is the name of the UPPAAL project to be generated: PV tool will generate <projectName>.xml and <projectName>.q files.

Javadoc Annotations for Process

The methods main() represent first of all the processes and process instances. The annotations respect Javadoc syntax and they are composed of two parts (see Figure 6-5):

1. One description sentence (line 12),
2. One custom Javadoc tag with its value (line 16),

```
11 /**
12     * Description sentence.
13     *
14     * @param test Test parameter.
15     * @return test result.
16     * @customTag JMLExpression
17 */
18 int testMethod(int test) {
```

Figure 6-5: Overall syntax of annotations

The value coming after the Javadoc tag has JML syntax, which is set of attributes, separated by commas and finishing with semicolon. Each attribute has the form key=value. The strings are within quotes and the arrays are within parenthesis. An example can look as follows:
The annotation for a process has the custom tag `@process` and has the following keys:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@process</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of <code>main()</code> method</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Possible with <code>@threadMain</code></td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur once per Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Short name of the process</td>
<td>Same as Java class naming</td>
<td>Class name to which <code>main()</code> belongs</td>
</tr>
<tr>
<td>quantity</td>
<td>Quantity of process instances</td>
<td>Integer &gt;= 1</td>
<td>1</td>
</tr>
</tbody>
</table>

An example is as follows:

```java
    int testMethod(int test) {
```

Javadoc Annotations for Threads

Concerning threads, there are three types of threads considered:

1. Thread of `main()` method (note that `main()` represents both a process and a thread);
2. Thread initiated by `run()` method of `Runnable` Thread class;
3. Thread for handling remote RMI method (a servant thread), initiated by each remote call by RMI middleware;
In the first case, there is just one `main()` thread instance in one process instance. Moreover, no name or description is needed – the name and comment from the process are taken, as there is one-to-one relationship between the process and the main thread. Therefore, there is just one annotation key: `isLooping`, specifying if once the thread finished one cycle, it shall loop.

Table 6-2: Annotation `@threadMain`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@threadMain</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of <code>main()</code> method</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Possible with <code>@process</code></td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur once per Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isLooping</code></td>
<td>Whether the end state shall be connected to start state</td>
<td>`true</td>
<td>false`</td>
</tr>
</tbody>
</table>

The example below (Figure 6-8) illustrates the annotations:

```java
class ThreadExample {
    @process name = "Atm" quantity = 3;
    @threadMain isLooping = true;
    public static void main(String[] args) {
        ...
    }
}
```

In the second case, there can be several instances of the same thread within one process instance. Moreover, it is useful to give a short name to the thread. This results in the custom tag `@threadRunnable` and the following keys:
Table 6-3: Annotation @threadRunnable

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@threadRunnable</td>
</tr>
<tr>
<td>Location</td>
<td>In front of run() method of class implementing Runnable interface</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur once per Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Short name of the thread</td>
<td>Same as Java class naming</td>
<td>Class name of class to which run() belongs</td>
</tr>
<tr>
<td>quantity</td>
<td>Quantity of thread instances within one process instance</td>
<td>Integer &gt;= 1</td>
<td>1</td>
</tr>
<tr>
<td>isLooping</td>
<td>Whether the end state shall be connected to start state</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

The example below illustrates the annotations:

Figure 6-9: Thread annotations, type “Runnable” - example

```
15   /**
16   * Thread getting data from the server.
17   *
18   * @threadRunnable name = "Com", quantity = 2, isLooping = true;
19   */
20   public void run() {       
```

In the third case of threads handling RMI calls, the situation is more complex. The threads are not explicitly instantiated within the application. Rather, every time RMI middleware receives the request from the client, a thread is created on demand. As the tasks are created statically within UPPAAL, the user must estimate how many servant tasks are needed and specify this. There are two quantities to specify: number of servant threads per remote method (which is in worst case the number of clients using this method) and number of remote objects implementing the interface. For this, we use the custom tag @threadRemote with the following keys (the two quantities are the assumed to be the same for all the methods of the interface):
Table 6-4: Annotation `@threadRemote`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@threadRemote</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of the remote class.</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur once per Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>interfaceName</code></td>
<td>Shortened name of the Remote interface, which is used by clients (in lookup)</td>
<td>Same as Java interface naming</td>
<td>The remote interface name</td>
</tr>
<tr>
<td><code>quantityOfServantsPerMethod</code></td>
<td>Quantity of thread instances (servants) per each remote method and per one remote object</td>
<td>Integer &gt;= 1</td>
<td>1</td>
</tr>
<tr>
<td><code>quantityOfObjects</code></td>
<td>Quantity of objects implementing the remote interface within one process instance</td>
<td>Integer &gt;= 1</td>
<td>1</td>
</tr>
<tr>
<td><code>isLooping</code></td>
<td>Whether the end state shall be connected to start state</td>
<td>`true</td>
<td>false`</td>
</tr>
</tbody>
</table>

The related example is as follows.

Figure 6-10: Thread annotations, type “Remote” - example

```java
11                   * `@threadRemote`  interfaceName = "Account",  quantityOfServantsPerMethod = 3,
12  * quantityOfObjects = 2,  isLooping = true;
13       */
14       public class AccountImpl extends UnicastRemoteObject implements Account {

Figure 6-10 specifies that there are two objects of the class implementing the interface Account. Each method has 3 servant threads, therefore within one process instance there will be 2*3*2 = 12 thread instances for this interface. The number of threads gets quickly very high.
6.1.1.4 PV Output

In short, the processes, process instances, threads, thread instances are represented as UPPAAL templates and UPPAAL template instances.

Some main constraints of UPPAAL are that there are only one-dimensional arrays of template instances and that there can’t be structures of containing templates (but only basic data types). Moreover, it is difficult to give a unique ID for every instance if arrays of thread instances are used.

The output of PV tool is one XML file, defining the model, and one text file, defining the model properties. Using the two files, the UPPAAL tool verifies the properties against the model. The model has the following main parts: declarations, templates, instantiations and systems. The model and the properties are described in the next sections.

UPPAAL Declarations

UPPAAL declarations specify:
1. Global constants;
2. Global types (typedefs);

There is one global constant within the use case, called QUANTITY_OF_THREAD_INSTANCES, which is simply equal to number of all the instances of all the threads within the modeled system (setting a range for integer data types in UPPAAL is needed to avoid state explosion). Based on this, the data type Id is defined and is used to uniquely identify each thread instance. This gives (in our Atm-Bank test case there are 22 thread instances):

```
2 //identification of channels
3 const int QUANTITY_OF_THREAD_INSTANCES = 22;
4 typedef int [0, QUANTITY_OF_THREAD_INSTANCES - 1] Id;
```

Global variables are not used within this use case.

UPPAAL Templates

UPPAAL templates specify:
1. Timed automata using a graphical notation, and having XML as textual representation;
2. Local variables of threads in Java-like syntax,
3. Parameters (each thread instance at the creation is parameterized with its own values, enabling to differentiate between thread instances)

A timed automaton is made of locations connected with transitions. The location has following main properties:
1. An id (a string), used to uniquely identify the location within the whole model. It is not visible to the user and has a meaning of a primary key (to uniquely identify the node within the XML file);
2. A name (a string), unique within one thread, used to identify the location;
3. Set of invariants: conditions that are true within one condition;
4. Coordinates on the diagram;

5. Comment.

6. Flags:
   - Initial: whether the location is the initial one,
   - Urgent: time is not allowed to pass within it,
   - Committed: an urgent location with the constraint that an outgoing transition shall be executed before any other transitions.

A transition does not have an ID, nor a name. It has the following properties:

1. References to source and target locations (by the location id), which it is connecting;
2. Select: operation to select a random number within a range, used usually by the guard, synchronization or update as an array index;
3. Guard: condition that shall be true so that the transition can fire;
4. Synchronization: communication between two thread instances;
5. Update: post-synchronization operation, usually data update;
6. Coordinates on the diagram;
7. Comment.

In this use case, we address the issue of identifying the locations. A location has two identifiers: an id, and the name.

The id is an “internal”, meaningless primary key, which is global within the whole model. Every location created by PV tool has the id defined as follows: i dNNN, for example i d000, i d001, i d002. There is no constraints on ordering of the locations (locations that are “before” in the model may have a higher id).

The name is user-visible identifier, which shall be unique within one thread (different thread instances will have the same location names, but it is not a problem as they belong to different thread instances). This name is used among others in the definition of system properties. The name is defined as follows: l NNN, for example l 000, l 001, l 002. The number should start with 0 for each thread and increment along the template locations. Moreover, error locations (locations with mean that the system is in an error condition) shall have suffix Error, for example l 005Error.

Local variables are not used in this use case. Moreover, in this use cases, the timed automata are not internally analyzed – this will come in the next use cases, describing the communication. For now, the whole timed automaton is represented as one location, with attribute “initial” activated.

Due to the fact that there are only one-dimensional arrays, it is difficult to identify a specific thread instance (like processName_threadName_objName [procId] [threadId] [objId]). Using one-dimensional arrays makes it difficult for the user to identify the thread. On the other side, duplicating the templates (having the same template, representing the same thread, but within different processes), is maybe not very clean solution, but as the model is generated and is not supposed to be changed by hand, but enables to have a very simple model and is very clear to the user. At the end, we come up to the following list of templates:
### Table 6-5: Naming of threads

<table>
<thead>
<tr>
<th>Category of template</th>
<th>Naming</th>
</tr>
</thead>
<tbody>
<tr>
<td>For <code>main()</code> method</td>
<td><code>&lt;Process name&gt;&lt;NN&gt;</code>, where:</td>
</tr>
<tr>
<td></td>
<td>NN: 2-digit process Id</td>
</tr>
<tr>
<td></td>
<td>Process name: as specified by annotation</td>
</tr>
<tr>
<td>For <code>Runnable</code> threads</td>
<td><code>&lt;Process name&gt;&lt;NN&gt;&lt;Thread name&gt;</code>, where:</td>
</tr>
<tr>
<td></td>
<td>NN: 2-digit process Id</td>
</tr>
<tr>
<td></td>
<td>Process name and thread name: as specified by annotation</td>
</tr>
<tr>
<td>For <code>Remote</code> threads</td>
<td><code>&lt;Process name&gt;&lt;NN&gt;&lt;MM&gt;&lt;Method name&gt;</code> where:</td>
</tr>
<tr>
<td></td>
<td>NN: 2-digit process Id</td>
</tr>
<tr>
<td></td>
<td>Process name and thread name: as specified by annotation</td>
</tr>
<tr>
<td></td>
<td>MM: 2-digit id of the object implementing a remote interface</td>
</tr>
<tr>
<td></td>
<td>Note that the interface name is not specified (it is implicit, knowing the method name)</td>
</tr>
</tbody>
</table>

In ATM example, this results in the following threads (templates):

![Figure 6-12: Threads - example](image)

In the example shown by Figure 6-12, the four last lines define the bank process. There is one instance of Bank process (the first two digits, always 00), which has two Remote objects (the second two digits, 00 and 01), where each remote object has two remote methods.
Note that the process indexes are counted independently and start with 0. In the example, we have Atm00, Atm01, Atm02, Bank00. Two digits are reserved for process id, thread id within the process, object id within the process, as it is unrealistic from performance reasons to have more than 100 process instances\(^1\). The process names, thread names and method names follow the Java class naming (capitalization of the first letter).

Finally, each template may have several parameters, enabling to differentiate and customize thread instances of a given thread. Within this use case, we introduce one parameter, which will be used in the next use cases and which enables to uniquely identify each thread instance. As introduced in the previous section, there is the datatype \(\text{Id}\). Now, simply each thread has the parameter \(\text{Id}\) \(\text{me}\). When an instance is created (see next section), a given for the parameter \(\text{Id}\) \(\text{me}\) is set.

**UPPAAL Instantiations and Systems**

UPPAAL system declarations specify:

1. Instantiations of templates (i.e. instances of threads);
2. Specifications which of instantiations are within the modeled system (useful for testing, for having alternative systems to be selected).

To create thread instances (template instantiations) from threads (templates), the thread instances are declared in Java-like syntax. In case of main() threads, there is one-to-one mapping between the template and the system (as there is always one main thread instance in the process instance). However, for Runnable threads and Remote threads, there are several instances within process instance. Note that all thread instances for a given thread, within one process instance, differ only by parameters, which are passed at the instantiation.

<table>
<thead>
<tr>
<th>Category of thread</th>
<th>Declaration Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>main() threads</td>
<td>(&lt;\text{name}&gt; = \text{Name}{(\text{N})}; \text{where:} \text{N: process instance Id})</td>
</tr>
<tr>
<td>Runnable threads</td>
<td>As above</td>
</tr>
<tr>
<td>Remote threads</td>
<td>As above</td>
</tr>
</tbody>
</table>

\(^1\) One can ask why there are several templates for the “same threads”. For example Com thread exists as Atm00Com, Atm01Com, Atm02Com. This is because it is difficult to parameterize everything within the thread to have one thread supporting all related thread instances. The good compromise is to use instances if you have several thread instances of the same thread within one process only (e.g. there are several servants serving remote clients, all identical, within one server process).
The term `<Name>` in the above table represents each of the full template names (including all indexes). The thread instance name starts with small letter. The index N is a number given for each thread instance to uniquely identify it.

The corresponding example is as follows. There are 22 thread instances in our example, each with a unique id, given as template parameter:

![Figure 6-13: Creation of thread instances - example](image)

```java
//Creation of thread instances
qAtm00 = Atm00(0);
qAtm00Com00 = Atm00Com(1);
qAtm00Com01 = Atm00Com(2);
qAtm01 = Atm01(3);
qAtm01Com00 = Atm01Com(4);
qAtm01Com01 = Atm01Com(5);
qAtm02 = Atm01(6);
qAtm02Com00 = Atm01Com(7);
qAtm02Com01 = Atm01Com(8);
qBank00 = Bank00(9);
...
```

It is not enough just to create a thread instance. One needs to specify which thread instances are really considered by the model. For this, a simple Java-like declaration, which just lists all the thread instances, is used:

```java
system <list of thread instances separated by comma>;
```

The corresponding example is as follows:

![Figure 6-14: Definition of system as list of thread instances - example](image)
Once all the declarations are specified, for each of the thread instances one can see the resulting thread instances in the simulator window of UPPAAL, displayed as timed automata. The result in our example (for one thread instance only) could be as follows (note that for every thread instance there will be defined a timed automaton within the simulator window):

Figure 6-15: Resulting thread instance – example

```
@annotation gBank0000GetBalance00
```

**UPPAAL Properties**

No extensions to UPPAAL properties are foreseen in this use case.

### 6.1.1.5 PV Tool

**Parser**

The annotations (written as Javadoc comments) have a defined grammar using ANTLR [R9] grammar definition language (from the grammar definition, the code for the parser of comments is generated). The grammar definition for annotations (as just said - defined in ANTLR) needs to be extended to support the required new annotations, which are specified in section 6.1.1.3.

Note that apart from Javadoc annotations with @-prefix, Parser shall also read the first sentence of the pure Javadoc comment. To do so, the Parser just searches for the first dot in the comments of the analyzed Java statements.

The parser model defines various code entities defining various elements in the code, like “for loop”, “body of if conditional” etc. No extensions in the parser model are needed.

The overall syntax of PV Javadoc annotations are:

```
@annotation JMLExpression;
```

The `JMLExpression` can be either an atomic datatype (like integer, string) or a complex expression. The reason of applying this format is to be Javadoc-compatible – by defining custom Javadoc properties, they can be correctly parsed by Javadoc tool while generating Javadoc documentation and it can be correctly handled (formatted, highlighted etc) by Java IDEs. The `JMLExpression` is a set of attributes, separated by commas, where each attribute has the form `key = value`. The whole annotation therefore looks like follows:

```
@annotation key1=value1, key2 = "value2";
```
Building Generator Model by Generator

The Generator shall build the Generator Model (which is an intermediary abstraction of Java code), including the extensions introduced in this use case. The extended Generator logic is as follows. First, it searches for main methods, listed as the configuration parameter. For each identified method, the object of class MetaProcess (in short: MetaProcess object) is created, with the attributes with the values as set by the annotations (or with default values if annotations are not set).

Secondly, it searches for threads:

1. main threads: it simply for every Process object creates one MetaMainThread object;
2. Runnable thread: it searches void run() methods within classes implementing Runnable classes, for which there are start() invocations from identified processes (a sub-search is needed) and for each it creates RunnableThread object;
3. Runtime thread: it searches for every method within every Remote interface, which is remotely invoked within the identified processes (a sub-search is needed) and for each method it creates a RemoteMethodThread object and for each interface a RemoteInterface object.

Generator Model

The base Generator and Generator Model (as specified in chapter 4) are substantially extended to support this use case. In particular, there are supplementary classes in the generator model (threads, tread instances, process, and process instances). To facilitate understanding what is appended within a use case, since now all classes introduced within the use cases will have a “*” suffix within their name within the design document.
The whole system, i.e. the distributed application, is represented by the top-level class Model. Model is made of several process instances, which is expressed by the composition of class Process with its attribute `quantityOfInstances`. Process instance is made of several thread instances, which is expressed by the composition of class `ThreadNode`, with its method `calculateNumberOfInstances()`. There are three most important types of Java threads. They are modeled as children of class Thread. In particular, they overload the method `calculateNumberOfInstances()`, which return the total number of instances of the given thread within one process instance. Using `calculateNumberOfInstances()`, one can determine the quantity of thread instances for every thread within one process, which is needed when determining the sizes of thread instance arrays. In more details:

1. Thread of `main()` method: there is only one instance per process instance. This is represented by class `MainThreadNode`, with method `calculateNumberOfInstances()` returning always 1,

2. Thread initiated by `run()` method of `Runnable` class. This is represented by class `RunnableThreadNode`, with method `calculateNumberOfInstances()` returning the attribute `quantityOfObjects` (which value is equal to the annotation `@quantity`),
3. Thread for handling remote RMI method (a servant thread). This is represented by the classes `RemoteMethodThreadNode` and `RemoteInterface`. The method `calculateNumberOfInstances()` returns `quantityOfMethodServants * quantityOfObjects`.

There are three counters that are used to index the process thread instances and to index the location ids and location names. This is described in details in the section “Building of Final PV Output Model”.

**Building of Final PV Output Model**

The generation of final PV model generation is executed top-down, starting with method `transform()` of class Model (Figure 6-16) and recursive invocation of the `transform()` methods. Every Model Generator class in its `transform()` method, will generate its corresponding representation in the output model and will call `transform()` method of the classes it contains. All the `transform()` methods create new objects under `PropertyDocument` (for properties), `NtaDocument` (for templates) and under `DocumentCode` (for Java-like statements within the model). Once all the statements are transformed, in the last step the method `transform()` of class model calls the method `writeCodeToNtaDocument()`, which merges the expressions (by correctly adding “//” before comments, and by adding commas and newlines where applicable etc) and writing them into the fields of `NtaDocument` (fields declaration, instantiations, system), and fields of `Templates` (declaration, parameter).
There are three counters within the Generator Model, which are used by `transform()` methods (see Generator Model: Figure 6-16). They are:

1. **threadInstanceCounter**: used to index thread instances: every thread instance has an unique id, which is set by the current value of `threadInstanceCounter`;

2. **locationIdCounter**: used to index the ids of locations: every location has a globally unique, meaningless primary key of the format `idNNN`, where the number `NNN` is set to the current value of `locationIdCounter`;

3. **locationNameCounter**: used to index names of locations: every location has a unique name within one thread (template) of the format `lNNN`, where the number `NNN` is set to the current value of `locationNameCounter`.

To correctly manage the attribution of indexes for location names, location ids, template instances ids, there are three methods within the class `MetaThreadNode`: `acquireLocationId()`, `acquireLocationName()` and `acquireInstanceCounter()`. These methods increment the related counters and return the value to be used by the client (where the client is a class of Generator Model, for example `MethodNode`).
Final PV Output Model

The final PV Output model represents the data before the serialization to:

1. model of application,
2. properties of application.

The model of application is created by generator by the XML API (classes NtaDocument, Template, Transition, Location). However, all the Java-like declarations within the UPPAAL model are just Strings, not organized within XML structure. Similarly, all the properties are now written to a big string.

The transformation approach is that each Generator Model has a method transform(), which transforms this element into a corresponding elements in Output Model. The problem is that it is difficult to distribute the generation now, because it is difficult to coordinate the writing of Java-like declarations by independent Generator Model classes (now the generation is executed by a central loop by Generator). To solve this, we introduce supplementary class structure to represent the Java-like code within UPPAAL (the tree starting with class DocumentCode, see Figure 6-18).

Figure 6-18: Final PV Output Model

As a general rule, to keep all class diagrams simple, for every private attribute there are corresponding getter/setter methods.
Class `DocumentCode` represents the container class, an interface and handler for all Java-like UPPAAL expressions. It offers methods to create new Java-like statements (`add*()`). There are three classes that define global model information (`GlobalDeclaration`, `Instantiation`, `System`) and two that define information local to one template (`TemplateDeclaration` and `TemplateParameter`).

*DocumentCode* is composed of the following global elements:

1. Declaration: one declaration of a typedef, global variable, global constant,
2. Instantiation: one instantiation (thread instance) of timed automaton (thread) – either as one instance, or as an array of instances,
3. System: selection of o instances to be active within the system, which is just the name of an instance,

*DocumentCode* is composed of the following elements local to one instantiation (thread instance) – they are defined within one thread, but the real data is local to one thread instance:

1. `TemplateDeclaration`: one declaration within the template,
2. `TemplateParameter`: one parameter of the template.

*DocumentCode* has a method `writeCodeToNtaDocument()` which calls the method `toString()` in the correct order of each of the objects in all the collections within *DocumentCode*. It also puts some comments/separators between consecutive `toString()` calls.

`writeCodeToNtaDocument()` also “escapes” the results of `toString()` of the classes `GlobalDeclaration` and `TemplateDeclaration` so that the output can be stored within xml fields.

**Logging**

To support error and warning reporting in a consistent way, we introduce a logger (Log4j). The logger generates two files: `error.log` and `warning.log`. In case of an error, the PV tool stops/exits.

### 6.1.1.6 Errors and Warnings

The following conditions are the errors and cause the PV tool to stop the extraction and return error message:

1. If two threads have the same name (a thread name is either the name given by annotation or the class name if the annotation is not given);
2. If two processes have the same name;
3. If any of values given by annotation does not respect the specified constraints (for example: a negative number for a quantity)
4. Duplicated annotation for one element if not allowed;
5. If two methods have the same name within one process (no interface name is given to keep the names short).
6. If exclusive annotations occur (like `@threadMain` and `@threadRunnable`, or `@for*` transitions introduced in next use case) for the same Java statement;
7. If an annotation occur several times for one Java statement, if it is specified that it can occur only once.
8. If any key within an annotation is twice or more times;
9. If an obligatory key within an annotation is missing.

The following conditions are warnings and cause the PV tool to return the warning, but let the tool continue:

1. The identified first sentence of the plan Javadoc comment contains the character @. It probably means that either the dot is missing, or the whole Javadoc comment is missing. In this case, the comment is set to empty string.

**6.1.2 Functional Aspects of Java Code**

The previous use case defined how to convert Java threads and processes to several instances of UPPAAL threads (timed automata). It also introduced a mechanism and naming convention for ids and names of timed automata locations. However, the timed automata have not been specified: there are just one “initial location”, and the whole logic of threads is not represented.

In this use case we define how each Java thread is to be represented in the abstracted form as a time automaton. The elements that are covered are for example function calls, standard Java loops and exceptions. Note that modeling of communication is not covered – this is separately addressed in 6.2.

**6.1.2.1 Analysis**

Java code elements that are to be represented in UPPAAL model within timed automata can be classified as:

1. Loops: for, foreach, while, do/while,
2. Conditionals: if/elseif/else an switch/case,
3. Local method calls (including constructor calls),
4. Communication calls: API calls for exchanging information with communication peers (like RMI calls, sockets) – not covered in this use case,
5. Java expressions.

Within this use case (and the consecutive ones for communication), we specify how these classes of Java code elements are to be represented as “patterns” of timed automata locations connected with transitions.

There are two main functional requirements for this use case, and both are related to simplicity of the output model. Firstly, the output model shall be readable and understandable by developer and assessor/reviewer. A developer must understand how his code is modeled in UPPAAL to be able to find errors. An assessor/reviewer must be able to understand why the model is correct and how it is working.

Secondly, the output model must be small. Only what is needed for the automatic property verification shall be extracted – otherwise, due to non-linear model verification time, we would finish with state explosion.
There is also a major constraint on the model: it must represent abstracted/simplified view of the code due to the limited representation features of UPPAAL with comparison with Java. For example, there are no Strings, objects, collections: they need to be abstracted to integers, structs, or arrays.

**Approach**

To address the above requirements and constraints, we propose a solution where the extraction of the model is user-supported. The user influences the model extraction by putting appropriate annotations in Java source code. This has two aspects:

1. Selection of subset of Java code: the user selects which Java code elements are relevant for the model and are to be extracted by putting annotations;
2. Abstraction of Java code: the user annotates Java statements by specifying more abstract representation of these statements.

The Generator Model – as specified up to this use case - represents sufficiently the Java code elements. However, the transformation step needs to be much extended. The transformation rules are as follows:

1. The transformation of each thread is iterative, starting independently with each ThreadNode;
2. Each node transforms itself, appending connected locations and transitions to already existing final PV model part. Then it calls transform of all elements in its body;
3. All the statements that are not annotated are ignored during the transformation. Annotation data are used as UPPAAL pseudocode.
4. Simple constructs (like expressions) are mapped to UPPAAL locations;
5. Complex constructs (like loops) are mapped to UPPAAL locations connected by UPPAAL transitions.

**6.1.2.2 Test Case**

The test case contains the following elements:

1. Loops: `for, foreach, while, do/while`,
2. Conditionals `if/elseif/else and switch/case`
3. Java expressions.
4. Java arrays.

Methods are introduced in the next section, as they depend on classes. Due to the size of the example, only the extracts are displayed in the relevant places. The whole example is available in source code (see chapter 11).
6.1.2.3 PV Input and Output

Within this use case, we show the Java input along with UPPAAL output for different Java constructs. First we define some general rules of inclusion (how to select the statements to be extracted) and then we define the annotations for each Java statement type.

Selection of Java Elements to Include in UPPAAL Model

The building of UPPAAL model starts with an empty template (just with an initial location) on UPPAAL side, and with the beginning of a thread on Java side (which is either the `main()` method, `run()` method or a remote method). Only some parts of the Java code are relevant for the application property verification and only they are to be included. Therefore, we define how the annotations are used in this purpose.

Firstly, if a statement (within a method) does not have an annotation, it is ignored, as well as all its body(see Javadoc comments for example explanations):

```java
59     /**
60     * Excluded because there is no annotation.
61     */
62     excludedMethod();
```

```java
17     /**
18     * Included because there is an annotation.
19     *
20     * @declaration;
21     */
22     int param;
```

```java
24     /**
25     * The loop is entered, because there is the annotation.
26     *
27     * @loopIf;
28     */
29     if (param == 1) {
30     */
31     */
32     * This is included because there is annotation.
33     */
34     int someVariable = 0;
35     */
36     */
37     * This statement is excluded because it has no annotation.
38     */
39     throw new IllegalArgumentException("illegal parameter");
40     }
```
Inclusion of the body of Loops and Local Calls

There are following four “basic annotations” for basic Java expressions:

1. Annotation @declaration,
2. Annotation @transition.

For loops, conditionals and local method calls there are supplementary annotations:

1. Loops: @loopWhile, @loopDoWhile, @loopFor,
2. Conditionals: @if, @switchCase,
3. Local method invocations: @method.

In case loop annotations or method annotations are used, their body is modeled (as described in the previous section). However, loops and local method invocations can be prefixed with basic annotations (@declaration and @transition). In this case, the method or the loop is not entered, and the annotations @declaration and @transition mean the simplified representation of the loop or the method.

In the example below, if conditional is ignored, and the body is simplified.
This feature is more useful for methods, especially for API calls. A method is simply abstracted as a set of transitions and declarations. So, if there is a method with @transition or @declaration annotations but without @method annotation, then the method is modeled as set of given declarations and transitions:

```
78       /**
79       * @declaration expression = "int[0, 10] someVariable2";
80       * @transition update = "someVariable2 = 0;
81       */
82       includedMethodNotEntered();
```

### Simple Expressions

A Java expression is mapped to one or more UPPAAL transitions and to one or more local or global declarations. The UPPAAL transition represents some action done by the system, and a declaration represents a declaration of a global or local variable. The following examples explain how the Java is represented in UPPAAL. In this example, the expression is converted to a simplified UPPAAL expression (it is up to the developer to decide about the simplification):

```
28       /** @transition expression = "b += 2"; */
29       b = Math.abs(b) + 2;
```

Corresponding output is:

1. Transition with the update (b +=2), which is connected to the already existing location (displayed as a location shown partially, because it does not belong to the output),
2. Output location (named l004), which is needed so that following transitions could be appended (locations can be seen as connections between transitions).

```
Figure 6-26: Simple expression: output
```

Since now, when we show a half-circle of the location, we mean that the transition is connected to the “previous” location, but this previous location is not the part of the current output.
In this example, `double b` is represented as integer with limited range:

```
16       /**
17       * @declaration expression = "int[0, 10] b = 0";
16       * @transition update = "b = 0";
19       */
20    double b = 0;
```

The output is firstly the declaration of the variable:

```
Figure 6-27: Variable declaration and setting: input
```

```
Name: BasicTestOC Parameters:.getHost
int[0, 10] b;
```

And also a transition:

```
Figure 6-28: Variable declaration: output 1
```

```
Figure 6-29: Variable setting: output 2
```

In some cases, the Java code can be used in UPPAAL in the raw (unprocessed) form. In this case, the annotation `@transition` is used for transitions, without any attributes. Missing `update` key mean that it gets the value of the Java statement:

```
Figure 6-30: Variable setting with default annotation: input
```

```
31       /** @transition */
32    b++;;
```

The corresponding output:

```
Figure 6-31: Variable setting with default annotation: output
```

```
Figure 6-31: Variable setting with default annotation: output
```

```b++```

Similarly, any Java declaration can be used in UPPAAL in raw form. In this case, the annotation `@declaration` is used. Missing `expression` attribute means that the Java statement is directly used in the output model, for example:
The corresponding output is an UPPAAL declaration within the template:

```java
int i = 0;
```

**Conditional If**

There are following annotations for the conditional “if”:

1. @if,
2. @ifElseIf,
3. @ifElse.

The three annotations have the optional `guard` attribute. In case the key is not given, the conditional expression from the Java is directly taken. If it is given, the user can specify abstracted condition (like it is done for keys `expression` and `guard` for `@declaration` and `@transition`). Let’s see an example:

```java
/**
 * @loopIf guard = "ifValue == 1";
 */
if (ifValue == 1) {
    /** @transition: */
    ifResult = 10;
    /** @transition: */
    ifResult *= 20;
}
/**
 * @loopIfElseIf guard = "ifValue == 2"
 */
else if (ifValue == 2) {
    // empty condition
}
/**
 * @loopIfElse;
 */
else {
    /** @transition: */
    ifResult = 0;
}
```

The guards on annotation are identical to the Java conditions; therefore guard keys can be omitted:
Both loop examples are mapped to the same output. The circles represent locations and the lines represent the transitions (the small circles on the corners of transitions have no meaning to the model – they are just highlighted corners of the transition).

The locations and transitions that are generated by the loop itself (without its body and without the previous/consecutive elements) are highlighted in yellow.
In UPPAAL there is no “native” else condition. The else is done by negating all if/else if conditions (i.e. \(! (\text{condition1} \mid \text{condition2} \mid \ldots)\)).

Although some merging of locations is possible at this state (like merging of locations l002, l003 and l004), this is not done to have a simple transformation step within the PV tool. In another use case it is described how to optionally merge locations as post-processing of created timed automata.

The general approach for deciding whether to create or not a new location by the loop is the following: if there will be more than 1 outgoing transition from the location, then create a new location and rather than just append to existing location. In our example of for loop, the location l001 is created, because there are three outgoing location from this.

**While Loop**

The while loop has only one annotation: `@loopWhile`, with optional key `guard`, for example:

![Figure 6-36: Conditional if-output](image)
In this example, the guard can be omitted. The output is as follows:
The first Java expression \(j = 10\) is mapped to transition and state \(l_{000}\). The while loop (and all other loops) do no “reuse” the initial states, they create their own initial state to the input state. In this case, the beginning of the while loop \(l_{0001}\) is connected to \(l_{000}\) and these states are not merged. This is in line with the general adopted approach of element transformation: the element receives as input the input location to which it shall be connected, it transforms itself and returns its output location (to which following elements will be connected). The element can attach only one outgoing transition to the input location – that’s the reason why \(l_{000}\) and \(l_{001}\) are distinct.

Summarizing what has already been said: the locations \(l_{000}, l_{001}, l_{003}\) and \(l_{004}\) with their input transitions are generated by the Java expressions \(j = 10, j--\) and \(j += 3\). The remaining locations are transitions (apart from the external initial location to which the loop is connected to) is generated as a representation of the loop while.

**Do-While Loop**

The do/while loop is very similar to while loop. It has one annotation: @loopDoWhile, with optional key guard, for example:

```java
29     int j = 10;
30     /**
31         * @loopDoWhile;
32         */
33     do {
34         /**
35             * Simplified operation (Math.random().ignored.in.the.abstraction).
36             * @transition update = "d = 2 * d";
37             */
38         d = Math.random() + 0;
39         /*
40             * @transition;
41         */
42         j--;
43     } while (j > 0);
44     /** @transition; */
45     d = 0;
```

Figure 6-39: Loop do-while: input
The corresponding output is:

![Loop do-while: output](image)

The locations generated by the loop directly are only \( l003 \) and \( l005 \) – other locations are generated from basic Java expressions;

**For Loop**

It has one annotation: \(@loopFor\), with optional attributes `init`, `guard` and `next`, like in the example below (Figure 6-41). In case keys `init`, `guard` and `next` are not given, then the Java expressions from the Java code are taken:
In this example, the loop has two annotations (declaration and loopFor). The corresponding output is as follows:

**Figure 6-41: Loop for: input**

```c
17    d = 0;
18    /*
19    * @declaration expression = "int i";
20    * @loopFor init = "i = 0", guard = "i < 10", next = "i++";
21    */
22    for (int i = 0; i < 10; i++) {
23        /*
24        * @transition:
25        */
26        d = 2 * i + d;
27    }
28    /* @transition: */
29    d = 0;
```

**Figure 6-42: Loop for: output**
Conditional Switch

It has the following annotations:

1. \texttt{@switch} without keys,
2. \texttt{@switchCase} with \textit{obligatory} key guard,
3. \texttt{@switchDefault} without keys.

The “case” conditions need all to be translated by the developer into the correct UPPAAL condition, therefore the condition is obligatory.

```java
int result = 0;

/**
 * @loopSwitch
 * @loopSwitchCase guard = "condition == 0"; */

switch (condition) {
    case 0:
        System.out.println("Value 0");
        break;
    /** @loopSwitchCase guard = "condition == 1"; */
    case 1:
        /** @loopSwitchCase guard = "condition == 2"; */
        case 2:
            System.out.println("Value 1 or 2");
            /** @transition */
            result = 1;
            break;
    /** @loopSwitchDefault; */
    default:
        System.out.println("Value out of range");
        /** @transition */
        result = 2;
        break;
}
/** @transition; */
condition = 0;
```

The corresponding output:
Operations on Arrays of Primitive Types

UPPAAL offers arrays of integers with the size fixed at compile time. To represent them, no new annotations are needed.

```java
/* Create an array of 10 keys. */

/* Declaration */
expression = "int[0, 9] phoneKeys[10]"

/* transition */
.int[] phoneKeys = new int[10];

phoneKeys[0] = 0;
```

The corresponding output is the declaration:

```java
int[0, 9] phoneKeys[10];
```
Multi-dimensional arrays are not supported by UPPAAL and therefore they are not supported by PV tool.

**Random behavior**

There are cases when we assume random behavior:

1. When we don’t want to model some behavior precisely, and abstracting from this we can only say that the behavior is random (example: a function returns an integer, but the logic is quite complex so we assume that it returns a random value);

2. When the behavior itself is random (example: reading a pressed key from the keyboard).

To address this, UPPAAL has a “select”-key for a transition. Let’s see an example:

```plaintext
double d = (Math.random() * 100) + 7;
```

In UPPAAL pseudocode we need to introduce a supplementary variable (required by UPPAAL), \( i \), which is randomly set with the line \( i : \text{int}[0, 100] \). The output template is as follows:
Naming Clashes, Multiple Annotations, Order of Annotation

Firstly, in Java, there is advanced name context. There can be elements under the same name:

1. variables in loops of the same method,
2. variables in different methods,
3. attributes and method variables,
4. attributes of different classes.

In UPPAAL, there is only differentiation between global data (methods, variables) and local to each thread instance (template instance).

In order not to introduce heavy naming convention (like prefixes of function names or class names), we require that variables have unique names within thread instances.

Secondly, the annotations are resulting in their corresponding UPPAAL output in the order of execution of Java code. So, if one @transition is after another @transition in Java, then it is also after in UPPAAL model, in terms of runtime execution order.

Furthermore, the annotations @declaration and @transition can occur together with others at the same Java statement. There can be any quantity of annotations @declaration and @transition together with remaining annotations. In case there are @loop* annotations and @transition annotations together, the @transition before/after the @loop* will generate UPPAAL transition respectively before/after the WHOLE loop. This applies also to methods and communication calls (not only to loops), which will be introduced with next chapters. Let’s see an example:

Figure 6-50: Multiple annotations: input

```java
13     /**<
14         * @declaration expression = "int a";
15         * @transition update = "a = 0";
16         */
17     int a = 0;
18
19     /**<
20         * @declaration expression = "int i";
21         * @transition update = "i = 0";
22         * @LoopWhile;
23         * @transition update = "a++";
24         */
25     for (int i = 0; i < 10; i++) {
26         /**<
27         * @transition;
28         */
29         a += 1;
30     }
```
In the first block, the statement `int a = 0` is converted to two UPPAAL statements. In the second block, the `for` loop is converted to declaration + transition + loop with its internal contents + transition. In this case, appending the transition to the loop is an alternative for putting the annotation for the `a++` Java statement. The result is as follows:

**Figure 6-51: Multiple annotations: output**

![Diagram](image)

**Specification of Introduced Annotations**

After having presented all the annotations with different context usages, step by step, we now specify these annotations.

The annotation `@transition` can be in front of any statement within any Java method and is mapped to UPPAAL transition:
Table 6-7: Annotation \texttt{@transition}

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>\texttt{@transition}</td>
</tr>
<tr>
<td>Location</td>
<td>Any statement within a method</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>None</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur several times for one Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{select}</td>
<td>Selection of random value</td>
<td>Conformant UPPAAL Select expression</td>
<td>Empty string</td>
</tr>
<tr>
<td>\texttt{guard}</td>
<td>Unused, undefined</td>
<td>Unused, undefined</td>
<td>Empty string</td>
</tr>
<tr>
<td>\texttt{update}</td>
<td>Quantity of thread instances within one process instance</td>
<td>Integer $\geq 1$</td>
<td>Java expression to which the annotation is attached (regardless of the existence of other annotations for the Java statement)</td>
</tr>
<tr>
<td>\texttt{sync}</td>
<td>Synchronization (reception/send of a signal) from/to another template</td>
<td>Channel access expression</td>
<td>Empty string</td>
</tr>
</tbody>
</table>

\textit{select} (introduced in further use cases)
Table 6-8: Annotation `@declaration`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@declaration</code></td>
</tr>
<tr>
<td>Location</td>
<td>Any statement within a method</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>None</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur several times for one Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expression</code></td>
<td>Declaration of variable</td>
<td>Conformant UPPAAL declaration.</td>
<td>Java expression to which the annotation is attached (regardless of the existence of other annotations for the Java statement)</td>
</tr>
<tr>
<td><code>isGlobal</code></td>
<td>Whether it is declared within the template, or globally</td>
<td><code>false</code></td>
<td><code>false</code></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Annotation</td>
<td>@loopFor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Location</td>
<td>In front of for loop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Remaining annotations of the statement can be @transition and @declaration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one @loopFor annotation per Java statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Key</strong></td>
<td><strong>Meaning</strong></td>
<td><strong>Value constraints</strong></td>
<td><strong>Assumed value if not specified</strong></td>
</tr>
<tr>
<td>init</td>
<td>Initialization expression in the loop</td>
<td>Conformant UPPAAL update expression.</td>
<td>Initialization statement within Java for loop</td>
</tr>
<tr>
<td>guard</td>
<td>Guard condition for the loop</td>
<td>Conformant UPPAAL guard expression.</td>
<td>Guard condition within Java loop</td>
</tr>
<tr>
<td>update</td>
<td>Update (counter incrementation) expression in the loop</td>
<td>Conformant UPPAAL update expression.</td>
<td>Guard condition within Java loop</td>
</tr>
</tbody>
</table>
### Table 6-10: Annotations \@loopWhile, \@loopDoWhile

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@loopWhile, @loopDoWhile</td>
</tr>
<tr>
<td>Location</td>
<td>For both loop types, in front of <code>while()</code> statement</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Remaining annotations of the statement can be @transition and @declaration.</td>
</tr>
</tbody>
</table>
| Multiplicity   | There can be only one \@loopWhile annotation per Java statement  
|                | There can be only one \@loopDoWhile annotation per Java statement |
| Key            | Meaning                                           | Value constraints            | Assumed value if not specified |
| guard          | Guard condition for the loop                     | Conformant UPPAAL guard expression. | Guard condition within Java loop |

### Table 6-11: Annotations \@if, \@ifElseIf, \@ifElse

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@if, @ifElseIf, @ifElse</td>
</tr>
<tr>
<td>Location</td>
<td>In front of statements: if, else if and else respectively</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Remaining annotations of the statement can be @transition and @declaration.</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one @if, @ifElseIf, @ifElse annotation per Java statement</td>
</tr>
<tr>
<td>Key</td>
<td>Meaning</td>
</tr>
<tr>
<td>guard</td>
<td>Guard condition for the loop</td>
</tr>
</tbody>
</table>
### Table 6-12: Annotations \@loopSwitch, \@loopSwitchCase

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@loopSwitch, @loopSwitchCase</td>
</tr>
<tr>
<td>Location</td>
<td>In front of statements: if, else if and else respectively</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Remaining annotations of the statement can be @transition and @declaration.</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one @if, @ifElseIf, @ifElse annotation per Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>guard</td>
<td>Guard condition for the loop</td>
<td>Conformant UPPAAL guard expression.</td>
<td>Guard condition within Java loop</td>
</tr>
</tbody>
</table>

### 6.1.2.4 PV Tool

**Building Parser Model by Parser**

The parser creates the intermediary parser model, as introduced in chapter 4. In this use case, we have seen that:

1. There are several different annotations (like \@transition, \@loopFor)
2. Each annotation may have several keys (like init = “i = 0”)
3. There can be several annotations per one Java statement,
4. Some annotations can be only for specific Java statements (like \@process for main() method signature) and others can be anywhere within a method (like \@transition).

To sum up, annotations are rather loosely related to a particular Java statement. To have a clean view and little code duplication, we introduce a hierarchy of annotations. Below are most important aspects of annotations:

1. **MCStatement** is an abstract class, for each Java code statement;
2. Each element from Generator Model references a Java statement from Parser Model, by the reference from Element to MCStatement;
3. There is an abstract class Annotation representing one annotation of one Java statement. There can be several annotations per one statement;
4. For each annotation type (like \@transition) there is a corresponding class (like TransitionAnnotation, ForAnnotation shown on the diagram);
5. `MCStatement` has the method `analyzeComents` which splits the comments by the delimiters “@” and “;”, and for each creates the corresponding Annotation object;

6. Each child of class `Annotation` implements the `analyzeAnnotation()` method, which creates an ANTLR parser for its `annotationString` and sets the keys; For example, the `ForAnnotation` parses `annotationString` and sets the values `guard`, `init`, `next`.

7. The parsed values of child classes of `Annotation` (like `guard`, `init` next of `ForAnnotation`) are accessible using getter methods, from classes of `GeneratorModel`. For example, `ForNode` accesses `ForAnnotation` through `Element`'s attribute pointing to `MCStatement`'s list of Annotations containing `ForAnnotation`.

Figure 6-52: Parser model extensions to support annotations

### Generator

The initial work done by Generator is described in the previous use case (section 6.1.1.5). In that use case, only generation of elements representing processes and threads is done. The new functionality introduced here is the building of the Generator Model of the code defining the functional aspects of the Java code. The code conforms to the Generator Metamodel specified in chapter 4.

### Generator Model

The elements of the code, built up by the Generator, form a hierarchy of elements. For example, a for-loop contains its body.

For example, a for-loop (a composite) contains as its elements the for-loop conditions, as well as one element for each Java statement within its body.
Building Final PV Model

The generation of final PV has been introduced in the previous use case (section 6.1.1.5), where only the templates and template instances are created. In this use case, the whole “body” (i.e. locations and transitions) of timed automata are to be created.

Due to the fact that each parent element within the Generator Model is a composite and it contains model elements that are invoked within this parent element, it is enough to invoke transformation by thread start methods and then the final output model is generated iteratively, in a top-down approach.

For this, we introduce transform() method. Each element has a full knowledge and context information to generate correct output representation (as part of output model – transitions and locations) for itself and for elements called within it.

```
/**
 * Transforms the element to output PV model, including all elements inside
 * this element.
 *
 * @param ntdoc Reference to whole XML document before serialization.
 * @param template Reference to related format (null if not applicable).
 * @param propertyDocument Reference to representation of UPPAAL property
 * document before serialization.
 * @param documentCode Reference to representation of java-like UPPAAL
 * element.
 * @param threadNode Reference to threadNode of the thread of the
 * element.
 * @param inputLocation Location to which the transformed element shall
 * attach itself.
 *
 * @return Output location of the element – there is always only one.
 */

public abstract LocationDocument.Location transform(NtaDocument ntdoc,
TemplateDocument.Template template, PropertyDocument propertyDocument,
DocumentCode documentCode, ThreadNode threadNode,
LocationDocument.Location inputLocation);
```

In particular, the element has access to the following elements introduced in the 1st use case: property document, current template, document code.

The element has also as parameter the reference to the location to which it should attach its representation (together with all contained elements). The method returns its final location. Due to the iterative/hierarchical structure, each element has one input location and one output location.

A very important aspect is the definition of default behavior in case the annotations or attributes of annotations are not defined. Intuitive default values enable developer save much time and reduce the synchronization need. There is a major difference between.

1. If a value of an attribute is an empty string (example: @transition guard = " "); this means that the developer explicitly sets this to empty value. This makes sense if there are other attributes specified for the annotation;
2. If an attribute is not present (example: @transition): this means that the default value for the attribute shall be taken according to the rules defined for the given annotation.

We don’t want to return null values, therefore:

1. If the key is not present, the getter returns Annotation.DEFAULT,
2. If the key is empty, then simply an empty string is returned.

The annotations themselves do not set the default values. Instead the nodes (from Generator Model) try to read the annotation keys (by calling for example getGuard(), getInit(), getNext() of ForAnnotation). In case the return is Annotation.DEFAULT, the value from Java statement is used.

Let’s see an example – a sequence of transformation of ForNode with its annotations to final PV output.

Figure 6-54: Example transformation (for node)
6.1.2.5 Errors and Warnings

No new error and warning types – the previous use case(s) cover already possible errors in annotations. However, new annotations are introduced, with their rules for exclusiveness and multiplicity.

6.1.3 Object-Oriented Aspects of Java Code

Java is object-oriented, whereas UPPAAL is a simple functional language, even without pointers. We propose here an approach for representing Java concept in a simplified way in UPPAAL.

In short, we map classes to structs. Attributes are mapped to struct attributes. However, methods almost disappear in the timed automata.

6.1.3.1 Analysis

UPPAAL does not have a graphical representation for methods (methods exist only in pseudocode). In the PV tool, we decided to use UPPAAL pseudocode (and therefore functions) to express algorithms only. However, the chain of Java method calls needs to be at least “scanned” through the PV tool and expressed as appropriate by the templates.

Moreover, there are no object-oriented concepts (classes, objects) in UPPAAL. Therefore, we propose a solution to express the classes using UPPAAL structs, though it is a simplification. We do cover inheritance and polymorphism to a limited extend.

Finally, there is problem with the representation of methods by transitions and locations, especially the parameter and return value passing. For this, the primitive data types are copied, but for complex data types we need to use initial object names within the methods, because there are no pointers in UPPAAL.

Several options have been found how to model Java methods in UPPAAL. All of them have some disadvantages, which is mostly due to the limited feature set of UPPAAL. We propose here one solution that sounds the best.

There is no concept of methods/functions within timed automata. As previously stated, there are pseudocode methods, but they have only limited use – we use them to represent algorithms that are just support the modeling but are not under the verification.

Every time a Java method that is to be modeled in UPPAAL timed automaton is invoked, it needs to be “reincluded” within the timed automaton.

The parameters and return values are passed in two ways in the methods:

1. By copy (by copying of the element) – for primitive data types like int, double;
2. By reference (by copying the reference to the element) – for arrays and objects.

Method parameters have usually different names than the names of the variables of the caller provided to the function. Moreover, a given method can be called in different places with different variables as parameters.

To solve this, for primitive data types, we introduce optional explicit copy of caller’s variable to the method’s parameter variable. The same is done for primitive return values.
However, for arrays and objects, it is only the reference that is copied. There is no straightforward way to represent reference copying in UPPAAL (references/pointers can be used only in the template instantiation) and copying data is too far semantically from Java. Therefore, for arrays and objects, the method call is “transparent”: within the method, the object is shown under the caller’s object name. Fortunately, every method invocation generates a separate transitions and locations, so every method invocation will use the initial object names of its caller.

To support methods, the annotations are in three places:

1. At the signature/declaration of the method – annotation @method and @methodNew (the first is for normal methods, the latter is for constructors),
2. At the invocation of the method - annotation @call and @callNew,
3. At the return line within a method – annotation @return,
4. At every copying of a reference to an existing object – annotation @newReference.

6.1.3.2 Test Case

The test case Java program contains the following elements:

1. Classes, interfaces, objects,
2. Static and non-static methods and attributes,
3. Method calls with parameters and return values as primitive and complex data types,
4. Constructor calls,
5. Exception throwing,
6. “External” classes, for which source code is not available,
7. Class inheritance,
8. Arrays of objects.

Moreover, we create objects within an array in order to demonstrate that the objects are not “colliding” with each other.

6.1.3.3 PV Input and Output

Within this use case, there are several Java code features that are presented. Each input is shown together with the corresponding output.

Classes

We need to represent classes, because they store data that may be relevant for modeling. We simply map classes to UPPAAL structs.
Figure 6-55: Class: input

```java
 public class Circle {

  /**
   * @attribute
   * Radius of the circle. Primitive data type: no problem to model this.
   */
  private double _radius;

  /**
   * Sorry, UPPAAL has no pointers. To identify referenced object, the name
   * used at the creation is used. This may change when UPPAAL is extended in
   * future.
   */
  private Point _center;

  /**
   * Color of the rectangle (red, green, blue) – 3 element array. Again, it is
   * an array so it is ignored.
   */
  private int[] _rgbColor;

  /**
   * Default radius of a point.
   */
  private static int DEFAULT_RADIUS = 10;

  /**
   * No.FV.annotation -> ignored.
   */
  private int someInternals;

  The attributes that are not annotated are ignored. This code maps to the following UPPAAL
  struct and to constants, both within global declaration context.
```
// Global declarations

/**
 * Circle, with its center and diameter.
 */
typedef struct {

    /**
     * Radius of the circle.
     */
    int radius;
}
} Circle;

const int CIRCLE_DEFAULT_RADIUS = 10;

We always use typedefs as well as structs without name, so the format is always:
typedef struct { ... } Name;

Classes from External Libraries

In case the source code of a class is not available (in case of external libraries), we can annotate the import statement. Here, we simply list the attributes of the imported class, and for static constants we explicitly define an expression, prefixed with class name:

Figure 6-57: Class from external library: input

```c
/* @classImport name = "AwtPoint", attributeDeclarations = {"int x", "int y"};
 * @declaration expression = "const int AWTPOINT_DEFAULT_X = 0";
 */
import java.awt.Point;
```

This is mapped to:

Figure 6-58: Class from external library: output

```c
/**
 * External class java.awt.Point.
 */
typedef struct {

    int x;
    int y;
}
} AwtPoint;

const int AWTPOINT_DEFAULT_X;

Class Inheritance

To represent the inheritance, the annotated attributes from all parent classes are included.

```
 9  * @class ColoredPoint
10  */
11  public class ColoredPoint extends java.awt.Point {
12
13  /**<
14      * @attribute declaration = "int rgbColor"
15  */
16  private int rgbColor;
17
18 }
```

This maps to:

```
/**
 * External class java.awt.Point.
 */
typedef struct |

  /**< inherited */
  int x;
  /**< inherited */
  int y;

  /**<
   * 32-bit, 3st 3 bytes are used for 3 colors.
   */
  int rgbColor;

} ColoredPoint;
```

The attributes of java.awt.Point are included because at the import statement there is \@class Import annotation (see previous example). Otherwise, the attributes from parent classes are ignored, in the same way as all not annotated attributes.

Interfaces

Interfaces are not represented in UPPAAL model, and there is really no need for this.

Methods with Primitive Parameters and Return

The annotation \@method has the key parameters, which value is the abstracted signature (with possibly ignoring some parameters and simplifying data types).

The annotation \@call has the keys:
1. **parameters** – a string with expressions passed as parameters for the method (like in method signature; The order is relevant);

2. **return** – variable name receiving the return value (in case different than the one from Java statement).

Let’s see an example:

```java
38   * @declaration expression = "int x";
39   * @call return = "x";
40   */
41   double x = circle.getRadius();

42   /**
43   * @declaration expression = "int newRadius";
44   *
45   * @call parameters = "x - 1, x + 1", return = "newRadius";
46   * The two parameters are optional here – they can take default values.
47   */
48   double newRadius = circle.setRadiusRandomRange(x - 1, x + 1);

49   /***
50   * @call:
51   * The same as above, with default values.
52   */
53   newRadius = circle.setRadiusRandomRange(x - 1, x + 1);
```

The two calls of `setRadiusRandomRange` (apart from the initial `newRadius` declaration) generate the same output. In the first case the keys are given, in the second case the default values are taken.

On the other side, the method has an annotation when it is declared:

```java
137   * @method parameters = "int minRadius, int maxRadius"
138   */
139   public double setRadiusRandomRange(double minRadius, double maxRadius) {
140   /**
141   * @transition update = "this.radius = (minRadius + maxRadius) / 2";
142   * In this abstracted form, we just do an arithmetic average
143   * of integers.
144   */
145   _radius = minRadius + (maxRadius - minRadius) * Math.random();
146   /**
147   * @return expression = "this._radius"
148   * Default return expression.
149   */
150   return _radius;
```
The key parameters of @method specifies the abstracted signature. Of course, the order of declarations within the key is important.

The key @return with its key expression identifies and abstracts the return expression.

Behind the scenes, there is work done by the PV tool:

1. For each method invocation with @call annotation, the corresponding method is of the corresponding object is located according to Java behavior (including overloading, overriding, inheritance rules) – not according to annotation matching. This location of the corresponding method is done using the ANTLR tool;

2. The parameters within @call are matched to the parameters of method (one-to-one mapping, for example int minRadius is mapped to x - 1);

3. For all mappings of the parameters, there is a one transition created, which copies the data (example: “minRadius = x - 1, maxRadius = x + 1”);

4. Moreover, for every parameter within @call parameters string, the declaration is created (example: int minRadius) - but only once per method;

5. Then the code within the body of the method is parsed. The annotations can use any variables defined within the template (including within the callers), which is used for accessing the object instantiated by the caller;

6. When a method returns, there is an annotation @return with the key expression. For the copying of the return value, a new transition is created, where the expression of @return annotation is copied to variable specified in return key of @call annotation (example: newRadius = _radius);

The PV tool variable $(this)$ is used in this use case and is converted to circle. The variable as well as other PV tool variables are specified in 6.1.4. After this introduction, let’s see the output:
Let’s see some details:

1. Between l001 and l002 the parameters are copied (mapping from @call to @method);
2. Between l002 and l003 the parameters are used and the attribute of the object is set. The usage of $(this)$ and mapping to struct names is explained in the next use case;
3. Between l003 and l004 the return value is copied back.

**Handling of Complex Data Types - Constructors**

Complex data types (objects, arrays) used for parameters and return values are treated a bit differently when passed as parameters or return values. They are passed by reference (or saying differently, the reference to an object is copied). There is no way to represent reference/pointer copying in UPPAL\(^2\). Therefore, if within a method an object is used, it (i.e. the struct modeling the object) is identified with the name used at the creation of the object (at the constructor). Constructors are also specific, and are described in this section.

There three two cases when objects and its attributes are referenced:

---

\(^2\) Unless UPPAAL introduces pointers or references, which is not the case according to non-official messages on UPPAAL website.
1. When primitive attributes can be defined within the same class as the method itself: for this, variable \(\text{this}\) is used. \(\text{this}\) is replaced by the name by PV tool with the initial reference name.

2. The attributes can be defined in another class, which is passed by reference (parameter) to the method: to enable this, the method uses PV tool variables (in the form \(\{\text{referenceName}\}\)), where \text{referenceName} is the name of the corresponding parameter in the method signature.

3. Primitive attributes can be defined within the same class as the method itself: for this, also the \(\{\text{referenceName}\}\) variables are used.

UPPAAL tool replaces \(\text{this}\) and \(\{\text{referenceName}\}\) to corresponding names when the class is instantiated in the final PV model. Let’s see an example. First, we create an object, with its initial reference name. For this, annotation \text{@callNew} is used:

```java
Circle circle = new Circle(10, initialCenter);
```

The difference to the standard \text{@callNew} is that there is no return key. Instead, there is a very important expression key, from which the PV tool gets the initial reference name (circle in this case) and uses it to replace expressions \(\text{this}\) within the constructor. On the other side, there is annotation \text{@methodNew}, with the key parameters. The attribute parameters contains the list of primitive and complex parameters of the method.

```
public Circle(double radius, Point center) {
    /* @methodNew parameters = "int radius, Point center";
    */
```

Behind the scenes, there is work done by the PV tool:

1. For each method invocation with \text{@callNew} annotation, the corresponding constructor is located according to Java behavior (including overloading, overriding, inheritance rules) – not according to annotation matching. This location of the corresponding method is done using the ANTLR tool;
2. The parameters within @callNew are matched to the parameters of @methodNew (one-to-one mapping, for example `int radius` is mapped to 10);

3. For all mappings of the primitive parameters, there is a one transition created, which copies the data (example: “radius = 10”);

4. However, for complex parameters (Point center in our example), PV tool will replace every PV variable `$\text{referenceName}$` (`$\text{center}$` in our example) by the corresponding name from the @callNew (initialCenter in our example);

5. Moreover, for every parameter within @callNew parameters string, the declaration is created (example: `int radius`) – but only once per method.

6. Then the code within the body of the method is parsed. The annotations can use any variables defined within the template (including within the callers), which is used for accessing the object instantiated by the caller;

7. When the constructor returns, no further action is done.

The corresponding output:

Figure 6-66: Constructor call and implementation: output

![Diagram]

Let’s see some details:

1. Between l000 and l001 the primitive parameter is copied (mapping from @callNew to @methodNew);

2. Between l001 and l002 PV tool replaces `$\text{this}$` by the initial reference name (specified in `expression = "Circle circle"`), which is `circle`.

As you see, copying of the object references by parameter passing and by explicit expressions (example: `_center = center`) is not visible in the output. However, this copying of object references is used by UPPAAAL tool to get the initial object reference name and use it in the output.

**Handling of Complex Data Types – Methods**

Methods with primitive data types have already been introduced. Moreover, constructors with primitive and complex data types have also been introduced. Therefore, let’s see an example with further details of a method, and show also the use of default values.
We create a new point (with initial reference name `shiftedCenter`) and then pass this point to `circle`. As you see, all keys are optional and the default values are taken:

```java
35          /**
36          * @field newCenter;
37          */
38          Point shiftedCenter = new Point(300, 400);
39
40          /**
41          * @call;
42          */
43          circle.setCenter(shiftedCenter);
44
45          /**
46          * @call;
47          */
48          circle.move(500, 800);
```

The method `setCenter` is as follows:

```java
131 public void setCenter(Point newCenter) {
132     /**
133         * @newRef expression = "\$(this._center) - \$(newCenter)";
134         */
135         _center = newCenter;
136 }
```

The annotation `@newRef` only instructs the PV tool that there is another reference to the object created. As the attribute `_center` of Circle is a class, therefore it is also “virtual” – it is not represented in the output, but only internally in PV tool.

The method move is as follows (to keep it simple, the setters are represented as transitions, and not as methods):

```java
113          * @method parameters = "int x, int y";
114          */
115          public void move(double x, double y) {
116              /** @transition update = "\$(this._center)._x = x"; */
117              _center.setX(x);
118
119              /** @transition update = "\$(this._center)._y = y"; */
120              _center.setY(y);
121          }
122          }
```

In the method move, `\$(this._center)` is replaced by initial reference name, which is `shiftedCenter`. The output of the method `move()` is as follows:
Figure 6-70: Complex data types: output

Let’s see some details:

1. Between line 1000 and line 1001 the primitive parameters x and y are copied (mapping from @call to @method);
2. Between line 1001 and line 1002 the PV variable is replaced by the initial object reference and used within the expression;
3. Similarly between line 1002 and line 1003.

**Specification of Introduced Annotations**

After having presented all the object-related annotations with different context usages, step by step, we now specify these annotations.
### Table 6-13: Annotation `@class`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@class</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of class declaration</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Yes (i.e. no other PV tool annotations allowed).</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Alternative name</td>
<td>Same as Java class naming</td>
<td>Class name</td>
</tr>
</tbody>
</table>

### Table 6-14: Annotation `@classImport`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@classImport</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of class import statement.</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (for example, a <code>@declaration</code> is used for static attributes)</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Alternative name</td>
<td>Same as Java class naming</td>
<td>Class name</td>
</tr>
<tr>
<td>attribute declarations</td>
<td>List of primitive attributes with data types</td>
<td>List in the form “Type value”, like <code>{&quot;int x&quot;, &quot;int y&quot;}</code>.</td>
<td>Empty sting</td>
</tr>
</tbody>
</table>
Table 6-15: Annotation `@attribute`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@attribute</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of an attribute within a class</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Yes (i.e. no other PV tool annotations allowed).</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
</table>
| Declaration | Declaration of a primitive attribute | Conformant UPPAAL declaration. | Java expression to which the annotation is attached, but without the following strings (they are cut out by PV tool):  
- public/private/protected  
- static  
- transient  
- synchronized |
Table 6-16: Annotation `@method`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@method</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of an method within a class</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Yes (i.e. no other PV tool annotations allowed).</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parameters</code></td>
<td>List of parameters with data types</td>
<td>Like for Java signature.</td>
<td>The same as Java parameters signature</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Without parenthesis.</td>
<td>For example: “int x, int y”</td>
</tr>
</tbody>
</table>

Table 6-17: Annotation `@return`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@return</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of return expression.</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Yes (i.e. no other PV tool annotations allowed).</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expression</code></td>
<td>Expression passed to return, copied on the return value of the caller.</td>
<td>Like for Java expression</td>
<td>The value of the return expression (without return keyword).</td>
</tr>
</tbody>
</table>

Table 6-18: Annotation `@methodNew`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@methodNew</code></td>
</tr>
</tbody>
</table>
| Location  | In front of a constructor.  
|           | Shall not be used in front of object factory. |
| Exclusivity | Yes (i.e. no other PV tool annotations allowed). |
| Multiplicity | There can be only one annotation per Java class declaration. |

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
</table>
| parameters | List of parameters with data types | Like for Java signature.  
Without parenthesis. | The same as Java parameters signature  
For example: “int x, int y” |

Table 6-19: Annotation `@call`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@call</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of invocation/call of a method</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (before and after there can be transitions, declarations.</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
</table>
| parameters | Expression passed to return, copied on the return value of the caller. | Like for Java expression | Java signature,  
For example “int x, int y” |
| return | Variable name (without data type) receiving the return value of the method call | Like for variable/attribute naming. | Java variable name (without data type) |
### Table 6-20: Annotation @callNew

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@callNew</td>
</tr>
<tr>
<td>Location</td>
<td>In front of invocation/call of a constructor</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (before and after there can be transitions, declarations.</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>Expression passed to return, copied on the return value of the caller.</td>
<td>Like for Java expression</td>
<td>Java signature, For example “int x, int y”</td>
</tr>
<tr>
<td>expression</td>
<td>Class name and initial object reference</td>
<td>Naming like for classes and object references</td>
<td>Java expression For example: Circle c = new Circle() Has default expression Circle c</td>
</tr>
</tbody>
</table>

### Table 6-21: Annotation @newRef

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@newRef</td>
</tr>
<tr>
<td>Location</td>
<td>In front of a Java expression when object reference is copied.</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Expression copying</td>
<td>Expression using PV tool variables (like $(this) etc.</td>
<td>Error, no default value.</td>
</tr>
</tbody>
</table>
6.1.3.4 PV Tool

Most of the architectural elements to support this use case have already been introduced. The most important extensions are the related to handling of object references. PV tool generator shall store for each object all the reference names. These reference names are created:

1. when a method is called and an object is passed by reference,
2. when there is an explicit reference copy (annotated with @newRef).

If an object reference is used within annotation, it is always used as PV variable. Examples are: $(this), $(this._circle). When generating the final PV model, the tool should go the whole way back to determine the initial reference name (i.e. the reference name used when the object was constructed). For each annotation, there is a corresponding class (like MethodNewAnnotation for @methodNew).

6.1.3.5 Errors and Warnings

The following conditions are the errors and cause the PV tool stop the extraction and return error message:

1. Incorrect parameters of @call with respect to @method,
2. Incorrect use or use of inexistent PV variables,
3. Missing expression in @newRef annotation (this annotation has no default value).

6.1.4 Parseable Annotation Variables

In order to provide sufficient flexibility and functionality in specifying annotation values, the values shall be parseable by PV tool. Within this use case, we introduce basic variables, which shall be replaced by PV tool with a corresponding location dependent value.

6.1.4.1 Analysis

In order to support upcoming use cases, there is a need to have some variables that are understandable by PV tool. PV tool, once found a variable within an annotation, shall replace it with the value.

The variable syntax is $(variableName). There are no constraints on naming. Moreover, there is no need to have any separator before or after the variable. Therefore, the following expressions are correct (examples):

1. $(this)._x → will generate myPoint._x,
2. $(listName)Add(1) → will generate waitingTrainsAdd(1),
3. $(listName)Add(me) → will generate waitingTrainsAdd(me).
### 6.1.4.2 PV Input and Output

Table 6-22: UPAAAL variables, part 1

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(this)$</td>
<td>The initial reference name of the object in the context</td>
<td>$(this)._x \rightarrow point._x$</td>
</tr>
<tr>
<td>$(here)$</td>
<td>Location name that will be in the place where the annotation <code>@assert</code> is present</td>
<td>$(here) \rightarrow Gate00.1007$</td>
</tr>
<tr>
<td>$(variable\ name\ from\ parameter)$</td>
<td>Variable name passed as a parameter of a method. This will be replaced by the original reference name.</td>
<td>$(point)._x \rightarrow destination._x$</td>
</tr>
<tr>
<td>$(process)$</td>
<td>Process name $(processId)$</td>
<td>See below</td>
</tr>
<tr>
<td>$(processId)$</td>
<td>Process id</td>
<td>$(process)$(processId) \rightarrow Atm00 (main tread of Atm)</td>
</tr>
<tr>
<td>$(thread)$</td>
<td>Thread name</td>
<td>See below</td>
</tr>
<tr>
<td>$(threadId)$</td>
<td>Thread id</td>
<td>See below</td>
</tr>
<tr>
<td>Variable name</td>
<td>Meaning</td>
<td>Example</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>$(objectId)$</td>
<td>Object id (for Remote)</td>
<td>See below</td>
</tr>
<tr>
<td>$(method)$</td>
<td>Method name, for Remote methods</td>
<td>See below</td>
</tr>
<tr>
<td>$(template)$</td>
<td>Full template name</td>
<td>$(template)$ is equivalent to:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(process)$(processId) – for main threads</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(process)$(processId)$(thread) – for Runnable threads</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(process)$(processId)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(objectId)$(method) – for Remote threads</td>
</tr>
<tr>
<td>$(instance)$</td>
<td>Name of the instance.</td>
<td>$(instance)$ is equivalent to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>g$(template)$(threadId)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(template) Atm00Com</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$(instance) Atm00Com00</td>
</tr>
</tbody>
</table>

### 6.1.4.3 PV Tool

Further analysis is needed which PV tool subsystems need to be extended.

### 6.1.4.4 Errors and Warnings

The following conditions are the errors and cause the PV tool stop the extraction and return error message:

1. An unknown key is used – this may mean typing error.

### 6.1.5 Parseable PV Tool Pseudocode Libraries

This use case specifies how to define code libraries, which after parsing by PV tool, can be used as utility libraries by UPPAAL model.

### 6.1.5.1 Analysis

Graphical language scales with difficulties. Without keeping only the minimum information on the diagrams, they quickly become unreadable. Within the diagrams we want to show (simulate) only high-level behavior (like sending messages and some protocol-related logic). Details like algorithms are not useful.
On the other side, it is difficult to express even basic programs without having some typical algorithms / libraries (like management of FIFO queue). As we are not interested in showing the logic of these algorithms (we assume they have been tested and/or mathematically proven), we use the feature of UPPAAL pseudocode to express them.

Finally, it does not make sense to write a library / algorithm for each application. Therefore, we introduce in this case a functionality to have libraries of UPPAAL code (part of which are delivered with the tool, and others can be written by the tool user).

To provide high flexibility, reuse and simplicity for the user, the library code is parsed and only then it generates a correct UPPAAL code (where the parsing is just replacement of $variables by corresponding string/number values).

### 6.1.5.2 Test Case

We provide one library class, ArrayList, based on Java ArrayList. This is used within the example UPPAAL model. We simply add elements to the list within the for-loop and then remove an element from the middle and check the expected value and state of the list contents.

### 6.1.5.3 PV Input

The UPPAAL library is stored within PV tool source code tree, under the Java package de.fzi.final_pv.model.UPPAAL_library.

Within this package, we have the package tree that is one-to-one mapping from Java class structure (in most cases the same). The classes to be provided shall have the same names and have similar or the same method names.

The library class is very like a Java class:

1. Extension of the file is .u instead of .java,
2. The name of the file and the class is the same as the corresponding Java class,
3. The comments are conform to Javadoc standard,
4. The methods and variables are grouped within an artificial class (which has the same name as the file), which is removed by the PV tool,
5. Each class have some parametrizable values, which are given by the user as annotation data, like $(size),
6. To avoid name clashes and to have a possibility to have multiple instances, we prefix attributes and methods with a prefix of the object name.

The last point needs further details. We create libraries which aimed to be one-to-one mapping from Java libraries (like Collections). However, as there are no classes/objects in UPPAAL and we want to keep very similar API. We decided to prefix all attributes and methods with the object name. It means that if you have three instances of a list in the program, then you have three *add* methods in the program$^3$.

---

$^3$ An alternative would be to have a me/this pointer plus data structures, as one do object-oriented programming in C. However, this has a disadvantage that the APIs are different. Secondly, you can’t really pass the reference/pointer in UPPAAL.
Let’s take an example. Within the code, we need to:

1. Create the list,
2. Add elements,
3. Remove elements.

Let’s see how it happens. First, the list is created. For this, we need to specify the class name to use (ArrayListInteger) and specify the parameters (the name, which is the “object” name), the static list size (this is a simplification of Java code) and optional min and max allowed elements for the list.

The parameter classpath is optional. By default, the libraries are searched under de.fzi.model_generator.UPPAAL_library and then searched under classpath for user-provided libraries.

The adding and removing of elements is simpler, and it is just calling of the library method. The very important thing is that the value of the name key is added by user (clientIdsAdd() instead of add()):

```java
List<Integer> clientIds = new ArrayList<Integer>();
```
To use the list, there shall be provided an implementation of it. Here we show some extracts of `ArrayListInteger` implementation:
Figure 6-73: Library implementation: input

```java
class arraylistInt(String name, int size, int elementMin, int elementMax) {
    /**
     * Array of size+1 elements, each having the range minVal, maxVal. 
     * The first [] are for value ranges, the second [] are for array size.
     * For boundary checks, there is one supplementary element.
     */
    int[{$(elementMin), $(elementMax)}] $(listName)[$(size) + 1];
    /**
     * Index to the current top/last element on the list.
     */
    int[0, $(size)] $(listName)CurrentSize;
    /**
     * Adds an element to the end of the list.
     *
     * @param element an integer within the requested range to be added to the
     * end of the list.
     */
    void $(listName)Add(int[{$(elementMin), $(elementMax)}] element) {
        $(listName)[$(listName)CurrentSize++] = element;
    }
    /**
     * Removes an element from the end of the list, element an integer within
     * the requested range to be added to the end of the list.
     * @param index Index of the element to remove - shall be at currentSize.
     */
    int[{$(elementMin), $(elementMax)}] $(listName)Remove(int[0, $(size) - 1],
                    int[0, 100] element;
        int[0, 10] i;
        1 = index;
        $(listName)CurrentSize -= 1;
        element = $(listName)[index];
        // Shift all elements left (from N to N-1).
        while (i < $(listName)CurrentSize) {
            ...}
```
2. The parseable variables are not predefined: simply every non-standard key provided in annotation \texttt{@declaration}Library is searched within the library file and then replaced by its value;
3. The values provided for parseable variables can be either integers or strings.

6.1.5.4 PV Output

There are two main outputs when libraries are used:

1. For \texttt{@declaration}Library, there is the declaration of variables and methods within the template,
2. For calling of the library API, simply a \texttt{@transition} is used with the update statement, generating a transition on the template.

In the template library, the variables have been replaced by the values provided within the annotations (\texttt{listName}, \texttt{size}, \texttt{elementMin}, \texttt{elementMax}), here is an extract:

Figure 6-74: Library: output
At the end of the simulation, we see the correct “stack trace” (the values are “shifted” after 4\textsuperscript{th} element [after 16 there is 20], the 4\textsuperscript{th} element is correctly withdrawn and has value 18, there are 9 elements in the list (10 – 1 withdrawn):

![Figure 6-75: State of the library list](image)

6.1.5.5 PV Tool

Library

Most of the work to be provided is to create a set of common classes and algorithms that are supposed to be often used in models to verify. This includes:

1. Lists,
2. FIFO, LIFO queues,
3. Maps,

Parser Model

For every annotation type, there is a separate Annotation class (for example, for annotation \texttt{@transition} there is a class \texttt{TransitionAnnotation}). Such class is responsible for parsing the annotation (using ANTLR) and setting its attributes (for example, \texttt{TransitionAnnotations} has attributes select, update, guard, sync).

The same is applied for the newly introduced annotation \texttt{@declarationLibrary}. This annotation has some predefined keys (like \texttt{@class, @classpath}), but it has also \texttt{class-dependent} keys used to parameterize that class (for example, \texttt{ArrayListInteger} has keys like size, name).

Due to the fact that the class-dependent key is a priori not known (everyone can develop a library and change parameters, therefore the keys need to be \textit{discovered}. This is done by introducing the “signature” of the class, specifying the keys and their type. The type is either String, or integer.

More precisely, the parsing of the library class, done by method \texttt{analyseAnnotation()} of \texttt{DeclarationLibraryAnnotation} is as follows:

1. Read the keys \texttt{@class and @classpath};
2. Read the file specified by \texttt{@class and @classpath};
3. Search for the class signature: search for the first “(“ and “)” – the string within it is the signature (example: (int $size, String $name) )

4. For each key within the signature, replace the each occurrence of $key within the loaded library class, by the value given by the annotation.

5. Extract the string between the first “{“ and last “}”. Save it class as an attribute (like other annotation attributes, for example class, classpath), available with a getter. Then it can be accessed by a class of Generator Model, which simply gets the string and stores it in the TemplateDeclaration.

Building of Final PV Output Model

6.1.5.6 Errors and Warnings

The following conditions are the errors and cause the PV tool stop the extraction and return error message:

A key specified in the signature of a library class is not specified within the @declarationLibrary annotation (all the keys specified in the signature are obligatory, without default values).

6.2 Inter/Intra Process Communication

This group of use cases introduces the functionalities to specifically support and model various network communication patterns:

1. Remote method invocation,
2. Communication using message passing.

Use cases related to communication are very important. Within the scope of the thesis, only the most important aspects are covered. However, further analysis is needed.

UPPAAL has only basic synchronization as communication mechanism between templates: there is no data passing and there is no send-receive communication. The communication is done by synchronization point over a named channel: the sender sends a signal, and the receiver receives it. Using this and global variables, we model all types of communication.

The following example (Figure 6-76) demonstrates this in more details. gCaller00 sends the sync over channel gPermissionRequest (the sending is by putting character “!” after the channel name) and gGuardian00 receives the request over channel gPermissionRequest (the sending is by putting character “?” after the channel name)
6.2.1 Remote Method Invocation

RMI is the standard Java remote method invocation middleware technology. It is a software layer over TCP/IP sockets, which tunnels application method invocations to a generated communication protocol over TCP/IP.

At runtime, the client simply invokes the methods of a stub object, which represents one server object. The stub object communicates with the server middleware over TCP/IP. The server middleware then invokes a corresponding method of the remote object. For every remote method of remote object, the server middleware has a pool of thread instances.

Before the communication (remote method invocation) can start, the server needs to publish every remote object a naming registry, and the client needs to lookup the object at the registry. Only afterwards the client-server communication can start.

To keep the model simple, we ignore the communication with the infrastructure, in particular with the naming registry – we consider only application to application communication. This is in line with our overall approach: to model the application aspects/elements which really need to be modeled. In contrary, if there is a software bug related to initialization (communication with infrastructure), then this can be easily found by debugging.

Some examples covered by the described model are as follows:

1. Remote method invocation without considering parameters nor return values;
2. Invocation of methods of one of several remote objects implementing a remote interface;
3. Invocation of methods of particular server process instance (in case there are several servers);
4. Callbacks to client (i.e. separate remote method calls from server to client).

Note that remote invocation covering parameters and return values, as well as considering timing are covered in the consecutive use cases.
6.2.1.1 Analysis

The first major issue is that UPPAAL does only have basic synchronization as communication mechanisms between templates. In unicast case, there are two communicating templates, one is the sender and one is the receiver. The communication is done by synchronization point over a named channel: the sender sends a signal, and the receiver receives it. In order to simulate data passing (like parameters, return values, or payload in telegrams), global data is used. In order to model the remote method calls, we need to have two synchronizations: one for calling and one for returning of the remote method.

The second major issue is that it is difficult to associate the client with the server and on the other side be flexible and enable developer to specify associations.

Thirdly, there is no “automatic” (provided by UPPAAL) identification of the sender/receiver: server receives a function call request, but does not know who sent it and to whom to send it back.

A server with remote methods is modeled as independent thread instances, where:

1. For each process instances there are several remote objects,
2. Each remote object has several remote methods,
3. For each method, there are several thread instances.

There selection of the destination of a remote call is at four levels:

1. Selection of the process instance (example: there can be several RMI servers, the backup one is selected if the main one is not working);
2. Selection of object within the process instances (example: for every customer account there can be a separate object);
3. Selection of remote method within the object;
4. Selection of thread instance for executing the remote method.

Each level is differently selected:

1. Process instance is explicitly selected by caller, using an annotation matrix with id pairs;
2. In the same way (with annotation matrix) the object is selected;
3. Method is explicitly selected too, simply by method name matching;
4. The thread instance is randomly selected by UPPAAL (for each method there is a pool of thread instances).

This reflects the reality: the RMI caller selects the server, remote object and calls a specified method. However it does not select the thread instance. However, the server must send back the response exactly to the caller thread instance. This requires indexing of every thread instance and passing and storing this index by server.
It is enough have one identifier per one calling thread instance – there is no need for several identification points within one thread instance, because of sequential execution. If a thread has invoked a remote method, then it is waiting for the return for this method. When the server returns, then the client will be in the correct location. The global thread instance id has already been introduced in the first use case.

An UPPAAL transition takes place within allowed time frame (limited by optional timing constraints), which means that it is not deterministic. However, if there is a synchronization between two transitions (i.e. at one transition there is sending, and at the second one there is reception), then the two transitions are not interrupted by other transitions. Therefore, we use one global variable for specifying the currently calling client.

The complex issue is how to associate client and server thread instances. This is done by putting annotations – maps associating the servers to clients.

**Network Failure Modes**

We do not consider failure modes of RMI middleware and underlying network. Therefore, we assume that within the context of remote method invocation (client calls the server, the server returns to the client):

1. **Routing**: The data is correctly delivered from sender to receiver,
2. **Data corruption**: data is not corrupted/damaged/changed during the transfer,
3. **Order**: the data is not cached between the client and server, so there is no problem with message order,
4. **Timing**: the messages will be eventually delivered.

**Network Model**

Taking into account the considered failure modes (see above), we model the whole RMI communication as synchronizations between application thread instances, communicating over global data.

The major alternative would be to model the middleware itself (as supplementary set of thread instances). The RMI middleware could be written in UPPAAL code. However, you lose “visibility” of who is talking with whom and everything gets more complex. As long as RMI failures are not modeled, this is not needed, use cases like network delays can be modeled with direct associations.

**6.2.1.2 Test Case**

Within this use case, we show several small test cases, related to associations between clients and servers.

**6.2.1.3 PV Input and Output**

The biggest issue is how to associate clients with servers. This information is not fully available in the source code and must be provided by annotations.

Our approach is as always to have a client to server associations, through

1. explicit server process and server object selection at the lookup statement,
2. and by a “routing table”.

These two ways are introduced in the next two sections.

Note that this use case has a small error with respect to exchange of $gClient\_id$. This impacts the output representation: access to $gClient\_id$ should be the same like access to $gSender$ and route[] table in the next use case. Due to time constraints, the output diagrams have not been updated.

**Lookups**

Annotation `@rmiLookup` is used at the creation of the stub for the remote object. It determines the server process and server object to be accessed. Let’s see an example:

```
32 33 34 35 36 37 38 39 40 41 42 43 44 45
/**
 * @rmiLookup expression = "Account.account0", serverProcess = "0",
 * serverObject = "0";
 */
account0 = (Account) Naming.lookup("//127.0.0.1/account0");
```

The parameters of the keys can be even variables (like loop index, for example to call all server object one after another:

```
32 33 34 35 36 37 38 39 40 41 42 43 44 45
/**
 * @rmiLookup expression = "Account.account1",
 * serverProcess = "$i()", serverObject = "0";
 */
account1 = (Account) Naming.lookup("//127.0.0.1" + i);
```

Now, once PV tool knows how to associate the client with the corresponding server, the correct synchronizations between threads can be done.

The annotation `@rmiLookup` is similar to `@callNew`. As we never model the lookup internally, we treat it as a constructor. The key expression is used for the sole purpose to determine the initial object reference, so that it can be passed by parameter. However, this is only with the respect to the initial object reference. The data structure corresponding to the server object is not created here – it is created at the server.

Now we come to another important point of channels. Channels enable to exchange information between thread instances. In UPPAAL, There can be several listeners on the same channel, in this case the listener is chosen randomly. We use this feature to randomly select the thread serving the remote method. However, the remaining identification information (server process and server object) are not random: the client wants explicitly to talk with a particular object on particular server. Therefore, the channel naming for rmi calls is as follows:

$g<Server\_Process\_Name><process\_id><object\_id><Remote\_Method>Call$ – for calling and

$g<Server\_Process\_Name><process\_id><object\_id><Remote\_Method>Return$ – for return of remote method.
For example:

With `gBank0000TransferCall`, we invoke process Bank, process id 00, remote object id 00, and we call the method transfer (the letter g is for global variable).

Another important point is that the server needs to know to which client to return – it must return to the same client from which the call has been done. For this, the client (ATM in our example) sets the global value `gClientId` which is then read by the server thread at the synchronization and then saved. When returning, the client only receives on the return channel if the global value is set to its thread id, meaning that this return is for this client. This works because the synchronizations are atomic.

Finally, there are annotations `@call` and `@method`, which have already been introduced. We ignore first the parameters exchanged:

![Figure 6-79: RMI call: input](image)

This generates (on the client side):

![Figure 6-80: RMI call: output](image)

Some more details:

1. Between 1000 and 1001, ATM sends over channel `gBank0000TransferCall`, and sets global value `gClientId` with its id.

2. Between 1001 and 1002, ATM waits on signal on channel `gBank0000TransferReturn`, and receives if `gClientId` equals to its id.

At the server side, the situation is only a bit more complex (as a reminder, each remote method is modeled as independent thread). The method has annotation `@method`, which has already been introduced:
Figure 6-81: RMI declaration: input

The output is as follows:

Figure 6-82: RMI at server: output

1. Between l000 and l001, Bank receives over channel gBank0000TransferCall, and stores the global value gClientId within its local value;
2. Between l001 and l002, Bank executes the body of the remote method (not modeled);
3. Between l002 and l003, Bank sets the global variable gClientId, to inform that the return is for the thread with the specified id.
4. Between l003 and l004, the actual synchronization is done.

As you notice the state l003 is committed, meaning that the operations between l002 and l004 shall be atomic. This is necessary to avoid race conditions. It is important to note that the transitions l002-l003 and l003-l004 can’t be merged: the value shall be set before the synchronization, because for the synchronization to be done, the condition at the client shall already be true.
Finally, there is another important point. Of course, to have a client and a server communicating, the client and the server must have the same channel names, on which one sends and one receives. Some clarifications are needed for the remote interface and remote object, which is the part of the channel name:

1. On the server side, the annotation `@threadRemote`, which is in front of remote class declaration, has the key `interfaceName`,
2. On the client side,
   a. the annotation `@rmiLookup`, has the key expression, from which the interface name is determined.
   b. The annotation `@rmiRoute` has the key interface, from which the interface name is read directly.

Surely, the `interfaceName` (point 1) shall be identical to the one determined from expression (point 1) or interface (point 3).

**Client and Server Association – Routing Table**

Using `@rmiLookup` to specify the target is quite straightforward. However, it is not flexible enough, even if one uses variables for process and object ids. Moreover, there is much annotation duplication.

To address this, we introduce routing tables. A routing table is associated to one thread. It is specified in front of a thread (main thread, remote thread, `Runnable` thread), and it is made of `rmiRoutes` (with annotation `@rmiRoute`), where each route (line) specifies the inputs:

1. for a given remote interface,
2. for a given client process id,
3. for a given client thread id (if the client thread type is `Runnable`),
4. for a given client object id (if the client thread type is `Remote`),

And outputs:

1. use as target following server process id,
2. and use following server object id.

The points 5 and 7 are already given with `@lookup` annotation, but using the `@rmiRoutes`, more advanced expressions can be given. Moreover, default value can be given.

The general rule is as follows:

1. First, PV tool reads the server id and object id from lookup,
2. If not all are provided, determine the missing ones using the routing table. For each line:
   a. Check if for the current: (remote interface, client process id, client thread id), using the annotation the server process id and server object id can be determined.
   b. If yes, exit and use it.

Let’s see some examples:
In this example:

1. There are four Atm processes (annotation @process),
2. For the interface Account, the first two processes (0 and 1) communicate with server process number 0, and the last two (2 and 3) – with server process number 1. In both cases, they communicate with object id 0.

It can be that a Runnable thread calls a remote method. Now, depending on the Runnable thread id, another destination can be called:

In this example:

1. There are four Atm processes (not shown),
2. The first 2 Atm processes communicate with server process id 0 and :
   a. Thread 0 communicates with object id 0,
   b. Thread 1 communicates with object id 1,
3. Similarly, the last 2 Atm processes communicate with server process id 1 and :
   a. Thread 0 communicates with object id 0,
   b. Thread 1 communicates with object id 1,
In this example, keys serverProcess and serverObject are not given, therefore the routing table is analyzed and the correct output is selected (server process id, server object id).

In this example, the routing table is fully ignored:

```java
account0 = (Account) Naming.lookup("//127.0.0.1/account0");
```

In this example, the routing table is used partially. For the missing keys, the routing table is used. Because the routing table is used only to determine missing keys, one can consider it as “overriding” the routing table by providing a key, even it can be fully determined through the routing table:

```java
account1 = (Account) Naming.lookup("//127.0.0.1/account0");
```

This is an extremely interesting example:

1. The selection of the server id is from the routing table,
2. But the selection of object id is done within the loop.

The server id and object id are determined through the rmiLookup and routing table. The name of the interface to call is known from Java code (and the annotated name can be read by accessing the interface annotation). As a result, the channel names can be constructed.

The output is equivalent to the one presented in the previous section (with the exception of different ids), therefore it is not repeated.

**Remote Server Objects**

Up to now, we presented the remote methods as independent threads from each other. However, they do communicate: they share among other class attributes.

Fortunately, we have already a way to handle classes, there are only minor changes needed:

1. Annotation `@class` has new key: `isGlobal`, determining where the class is declared. For remote classes, the declaration must be global;
2. Annotation `@methodNew` has also key `isGlobal`, which is to be `true` for remote objects;
The point 2 causes a supplementary difficulty. It is enough to declare a global typedefs struct once, even there are several process instances, and there is no collision. However, he struct instances must be global because they are shared between the methods of one object. So for each server process id and object id, there is a separate struct generated:

![Figure 6-88: Construction of server remote object: input](image)

```java
16          * @callNew  expression = "Account.acc".isGlobal = true;
17          */
18          Account acc = new AccountImpl();
```

PV tool shall recognize that it is a remote interface, and therefore add a prefix for the initial reference name and capitalizes the provided reference. The prefix shall be as follows: 

g<ServerProcessName><processid><objectid>…

For example, we can get gBank0000Acc for the process id 0, object id 0.

Now, every time within any method of the class the object attribute is accessed with $(this)$, then $(this)$ is replaced with gBank0000Acc by PV tool (and not by acc as it would be for objects with isGlobal set to false).

**Passing of Application Data**

The passing of complex data types (arrays, objects) and their serialization is quite complex and is not covered in this use case. Here we propose just a solution for primitive data types.

To keep this intuitive, we apply the same approach as for local methods:

1. Annotation @call at the invocation of remote method,
2. Annotation @method at the declaration of remote method (in the class).

The only difference is that the data from the caller can’t be directly copied between the caller and the method – we must use intermediary global variables. Fortunately, as the transitions are atomic, therefore there is no problem with global variables. For each annotated parameter, we introduce a global variable, just capitalized and prefixed with g (example: amountInCents -> gAmountInCents).

Let’s see an example. Here we call the remote method (as the annotated parameter is the same as the Java parameter, it could be even omitted):

![Figure 6-89: RMI call with data passing: input](image)

```java
51          * @call parameters = "100";
52          */
53          account.transferAmount(100);
```

On the server side, we have a method declaration (within a remote class):

![Figure 6-90: RMI method implementation with data passing: input](image)

```java
27          * @method parameters = "int amountInCents";
28          */
29         public void transferAmount(int amountInCents) throws RemoteException {
```
In the local call, in the output we would just have the output:

1. declaration of local variable int amountInCents;
2. transition amountInCents = 100.

However, as the method is remote, we have:

1. declaration of local variable int amountInCents (as previously),
2. declaration of global variable int gAmountInCents (new),
3. At client’s transition, update gAmountInCents = 100,
4. At server’s transition, update amountInCents = gAmountInCents.

Let’s see the output templates. The client sets the global variable:

```
Figure 6-91: RMI call with data passing at client: output
```

```
1000

| gBank0000TransferCall |
| gClientId = me, gAmountInCents = 100 |
```

```
1001

| gClientId == me |
| gBank0000TransferRet? |
```

```
1002
```

The server gets the global variable and copies it on its parameters. Both setting by client and getting by server happen atomically (two synchronized transitions):

```
Figure 6-92: RMI call with data passing at server: output
```

```
1000

| gBank0000TransferCall? |
| clientId = gClientId, amountInCents = gAmountInCents |
```

```
1001
```

The same is used for the return values, using the `@return` annotation.
Specification of Introduced Annotations

The keywords have been introduced for annotations @class and @methodNew. Apart from this, two new annotations are introduced.

Table 6-24: Annotation @rmiLookup

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@rmiLookup</td>
</tr>
<tr>
<td>Location</td>
<td>In front of Naming.lookup()</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (before and after there can be transitions, declarations.)</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Class name and initial object reference</td>
<td>Naming like for classes and object references</td>
<td>Java expression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example: Circle c = Naming.lookup(s)</td>
<td>Circle c has default expression</td>
</tr>
</tbody>
</table>
<pre><code>          |                                           |                                                                                   |                                |
</code></pre>
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@rmiRoute</td>
</tr>
<tr>
<td>Location</td>
<td>In front of thread declaration</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (can be with @thread annotations)</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be several @rmiRoute annotations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Interface name for which the route applies</td>
<td>Existing annotated name</td>
<td>Error, it is obligatory</td>
</tr>
<tr>
<td>clientProcess</td>
<td>Id of the client process</td>
<td>Index(es) within the range, separated by comma</td>
<td>Error, it is obligatory</td>
</tr>
<tr>
<td>clientThread</td>
<td>Id of client thread, if client is a Runnable thread</td>
<td>Index(es) within the range, separated by comma</td>
<td>Obligatory for Runnable threads (error if not specified).</td>
</tr>
<tr>
<td>clientObject</td>
<td>Id of client object, if client is a remote thread</td>
<td>Index(es) within the range, separated by comma</td>
<td>Obligatory for Remote threads (error if not specified).</td>
</tr>
<tr>
<td>serverProcess</td>
<td>Id of the server process</td>
<td>Index within the range</td>
<td>Error, it is obligatory</td>
</tr>
<tr>
<td>serverObject</td>
<td>Id of the server object</td>
<td>Index within the range</td>
<td>Error, it is obligatory</td>
</tr>
</tbody>
</table>

6.2.1.4  PV Tool

Most of the features have already been introduced within the PV tool in the previous use cases, especially in the fist use case, and in the object-oriented aspect use case.

6.2.1.5  Errors and Warnings

The following new conditions are the errors and cause the PV tool stop the extraction and return error message:
1. Missing information to determine the routing.
6.2.2 Communication using Message Passing

The most popular communication pattern in high-integrity and embedded systems is message passing.

Within this chapter, we first do the classification of different networks, according to different aspects. Then, for most of the aspects we demonstrate how to model this.

Thanks to this approach, the developer can model very different networks, depending on the annotations he puts. Moreover, depending on his needs, he can model one particular Java network stack in different ways, depending on the particular needs.

6.2.2.1 Analysis

Let’s list the main aspects of network to classify them:

1. Addressing/identification: identification of messages with source/destination (source/destination id) or with message contents (message id),
2. Reliability (failure modes): detection/protection against message loss, corruption, sequence, duplication etc,
3. With Buffering of messages or without,
4. Synchronous/asynchronous data exchange,
5. Realtime or not (with bounded/guaranteed delivery time or without),
6. Unicast vs. multicast vs. broadcast,
7. Connection: connection-oriented or connectionless,

Each of the above aspects causes big differences in the representation of it in the output model. Within this use case, we will show several of the classes mentioned above.

6.2.2.2 Test Case

Within this use case, we show several small test cases, demonstrating following aspects and functionalities:

1. Addressing by sender and receiver,
2. Addressing by Message ID,
3. Hybrid addressing,
4. Data passing,
5. Communication timeout.
6.2.2.3 PV Input and Output

Addressing by Sender and Receiver

Most of computer networks deliver messages through the addressing. Each message has an id of the sender, and id of the receiver. The network stack routes the message to the receiver. Some examples of such networks are: Ethernet, IP, various fieldbus networks (PROFIBUS, PROFIsafe).

There are various addressing schemas in various networks. Sometimes it is a long number or a string and may have different representations. It can be composed of parts (for example, the socket address is composed of IP address and port number). Therefore, we abstract this and introduce a unique addressing schema: an integer identifying a sender address (or receiver address).

It is important to note that there is not necessarily one-to-one mapping between address and a thread instance. For example, one thread can listen on three different ports. But in any case, the addresses are to be provided by the developer.

Let’s see an example: in this case, we call some network function. We abstract this with the annotation. As source, the thread instance id is provided (me). Then, two receivers are notified:

```c
    /*
    * @send senderAddress = "me", receiverAddress = "1"
    */
    notifyReceiver(receiverAddress);

    /*
    * @send senderAddress = "me", receiverAddress = "2"
    */
    notifyReceiver(receiverAddress);
```

The receiver waits for the notification:

```c
    /*
    * @receive receiverAddress = "me"
    */
    getNotification();

    /*
    * @receive senderAddress = "0", receiverAddress = "me"
    */
    getNotificationFrom(0);
```

Here, the receiver first waits for any notification from any sender. In the second, it waits for the message, but only from the given sender. This enables to model sockets, when you configure if you accept messages from anyone, or only from restricted senders.
Note that “me” is an UPPAAL variable, therefore no $()$ is used (it is not interpreted by UPPAAL).

In general, the following holds:

1. There are several senders and receivers,
2. There can be several senders who try to send and receive at the same time (busy waiting),
3. There can be several senders for one receiver (many-to-one).

The following must be guaranteed:

1. The receiver must know from which sender it received the data,
2. The receiver must be able to receive application data (payload) from sender.

In particular, it may happen that there are several senders trying to send to one receiver at the same time. The receiver must accept one of them and must know which is selected. It poses very much difficulty, because this many-to-many communication must be done using shared global memory.

The logic at the sender is as follows:

1. Transition 1: The sender stores in a global array $gRoute$, under its index, the desired destination of the message. This array means that there is a pending synchronization for the sender and receiver,
2. Transition 2: The sender synchronizes with the receiver (network!), and afterwards it sets the global variable $gSender$, which means the actual sender on the current transition. Then it cleans the route.

The logic at the receiver is as follows:

1. Transition 1: The receiver synchronizes with the server (network?), but only there is any pending synchronization for this receiver (scanning the route[] table),
2. Transition 2: In the meantime, $gSender$ has been set by the sender. Now, the receiver copies the actual receiver from the global variable, over the committed location.
Figure 6-95: address-based send: output

And the receiver (in the first example, the receiver any sender, and in the second one, only from a specified sender):
At the sender side, we use a global utility method, which scans the `gRoute` table and returns `true` if there is any pending synchronization for that destination. The code is as follows:

Note that everyone can send and receive on the network in this model. There is just one channel called network. The recipient gets the data if its address is the same as the destination address of the message.
To sum up this section, this approach models well basic address-based unicast communication, like IP/TCP/UDP.

**Addressing by Message ID**

In some networks, there is no sender and receiver addressing: there is just a message id. Informally describing this, the sender says: “I have message of type id, receivers please take it if you are interested”. This is typically a broadcast network. In such a way works for example CAN network.

Let’s see how to annotate it: at the sender and receiver, we use the key `message`.

**Figure 6-98: message-based addressing send: input**

```
13        /**
14        * @send message = "gRebootNotification";
15        */
16        notifyAllReboot();
```

And the receiver:

**Figure 6-99: message-based addressing receive: input**

```
14        /**
15        * @receive message = "gRebootNotification";
16        */
17        waitForRebootNotification();
```

The corresponding output is the following:

**Figure 6-100: message-based addressing send: output**

```
gRebootNotification!
```

And on the receiver side:

**Figure 6-101: message-based addressing receive: output**

```
gRebootNotification?
```
There can be several receivers listening for message of type `gRebootNotification`.

**Hybrid Addressing**

The sender and receiver need to exchange the data (payload) send over the messages. This is explained in the next section (Data Passing).

Using data exchange, the sender and receiver can communicate. This happens over global data. However, instead of sending some identification data, the sender can name the type of the message, making the model more visible.

For example, instead of saying: “the first byte is the message type. If the type is 1, then it means synchronization”, we just give a distinct name for every message type.

Let’s see an example. Now, the channel is no longer called “network”, but for each message type there is a separate channel. The “sender” first sends the request (type `gReq`), and then waits for acknowledgement (type `gAck`):

```cpp
15      /**
16      * @send senderAddress = "me", receiverAddress = "5", message = "gReq";
17      */
18      sendRequest(receiverAddress);
19      
20      /**
21      * @receive senderAddress = "4", receiverAddress = "me",
22      * message = "gAck";
23      */
24      receiveAck(receiverAddress);
```

The “receiver” is as follows:

```cpp
13      /**
14      * @receive receiverAddress = "me", message = "gReq";
15      */
16      ReceiverC.getNotification();
17      
18      /**
19      * @send senderAddress = "me", receiverAddress = "4", message = "gAck";
20      */
21      ReceiverC.sendAck(4);
```
The output is as follows. The “sender” (note that the naming is a bit misleading, because the “sender” first sends and then receives too):

**Figure 6-104: Hybrid addressing send: output**

```plaintext
Figure 6-104: Hybrid addressing send: output

```

And the “receiver”:

**Figure 6-105: Hybrid addressing receive: output**

```plaintext
Figure 6-105: Hybrid addressing receive: output

```

**Data Passing**

The naming of channels in the hybrid solution, as introduced in the previous section, is a basic way of data exchange. However, modeling of exchange of further application data is needed.

We don’t need to understand how the information is (de)serialized – we assume that the peers exchange primitive data types. For this, we explicitly copy the transferred data from/to a global value, which by default carries the name of the receiver variable (plus prefix `g`):
The input is as follows: a sender sends to the receiver the speed. The key payload explicitly lists the copy expressions:

```c
15    /*
16        * send  senderAddress = "me", receiverAddress = "7",
17        * payload = "gReceivedSpeed = 100"
18        */
19        notifySpeed(receiverAddress, 100);
```

And the receiver gets this speed:

```c
18    /*
19        * receive receiverAddress = "<me>",
20        * payload = "receivedSpeed = gReceivedSpeed"
21        */
22        receivedSpeed = receiveSpeed();
```

Using the key payload:

1. The sender specifies copies the local variables or data on the global variables,
2. The receiver copies the global variables on its local ones.

The variables/expressions are separated by comma. Here we expect two integers to receive:

```c
24    /*
25        * receive receiverAddress = "<me>",
26        * payload = "receivedSpeed = gReceivedSpeed, recvTime = gRecvTime"
27        */
28        receivedSpeed = receiveSpeed();
```

The output is as follows (ignoring the last example with multiple data transmitted):
Note that all the channels, global variables introduced due to this use case need to be generated by the tool as well. The introduced declarations are simply not show because the declaration syntax is evident.

**Communication Timeout**

When receiving data, we need to be able to specify the reception timeout. In the receive annotation, we introduce keys timeout (specifying the time to wait until timeout) and key condition (a Boolean value set to `true` if the timeout happens). Let’s see an example:
The output is as follows:

1. The timer \( t \) is set to 0.
2. The receiver tries to synchronize within the allowed time (condition \( t \leq 10 \)).
3. If it manages, the condition \( \text{timeout} \) is set to \text{false}.
4. Otherwise (after the allowed wait time – condition \( t > 10 \)), then the timeout condition is set to \text{true} (\( \text{isTimeout} = \text{true} \)).

Then the timeout variable can be used in further parts of the code.
### Specification of Introduced Annotations

**Table 6-26: Annotation @send**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@send</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of any statement that sends data to remote thread</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>For message-addressed communication, the message id</td>
<td>Shall be the name of existing channel,</td>
<td>Value <code>gNetwork</code> → assumes using one common network and source-destination addressing. Example: <code>gAck</code>, <code>gNotifyCross</code></td>
</tr>
<tr>
<td>sender</td>
<td>Sender address</td>
<td>An integer</td>
<td>Error, obligatory</td>
</tr>
<tr>
<td>receiver</td>
<td>Receiver address</td>
<td>An integer</td>
<td>Error, obligatory</td>
</tr>
<tr>
<td>payload</td>
<td>Expression copying local variables on global ones</td>
<td>UPPAAL value copy expression</td>
<td>Empty string – assumed no application data passing. Example: <code>gPayload = payload</code></td>
</tr>
</tbody>
</table>
Table 6-27: Annotation \texttt{@receive}

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>\texttt{@receive}</td>
</tr>
<tr>
<td>Location</td>
<td>In front of any statement that receives data to remote thread</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>For message-addressed communication, the message id</td>
<td>Shall be the name of existing channel,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Value \texttt{gNetwork} $\rightarrow$ assumes using one common network and source-destination addressing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Example: \texttt{gAck}, \texttt{gNotifyCross}</td>
</tr>
<tr>
<td>sender</td>
<td>Sender address</td>
<td>An integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Empty string – means that a message from any sender is accepted.</td>
</tr>
<tr>
<td>receiver</td>
<td>Receiver address</td>
<td>An integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Error, obligatory</td>
</tr>
<tr>
<td>payload</td>
<td>Expression copying global variables on local ones, to represent data transfer</td>
<td>UPPAAL value copy expression</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Empty string – assumed no application data passing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Example: \texttt{payload} = \texttt{gPayload}</td>
</tr>
<tr>
<td>timeout</td>
<td>Time that is waited for the reception</td>
<td>A positive integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 - endless wait</td>
</tr>
<tr>
<td>condition</td>
<td>Variable getting true if there was a timeout</td>
<td>A Boolean variable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The timeout is still considered, but the output is not copied into the variable.</td>
</tr>
</tbody>
</table>

\textbf{6.2.2.4 PV Tool}

Most of the features have already been introduced within the PV tool in the previous use cases, especially in the fist use case, and in the object-oriented aspect use case. Two new annotations are introduced. One of tricky issues is how to differentiate between name clashes of global variables and usage of the same global variable.
6.2.2.5 Errors and Warnings

The following new conditions are the errors and cause the PV tool stop the extraction and return error message:

The reception payload does not correspond to send payload (different number of attributes);

6.3 Model Verification against Properties

Our approach is to provide to the developer the ability to specify UPPAAL properties together with Java source code, thus avoiding desynchronization problems. Moreover, as the properties are within the code, they are attached to some locations. Thanks to this, the developer can with ease specify location-dependent properties like “\texttt{A[] \\$_{\textit{here}}\texttt{ someCondition}”}. Moreover, using PV tool variables, the developer can create even sets of similar properties. Finally, some typical properties (deadlock-freeness and reachability) are by default generated for every state of all templates.

PV tool supports the developer in doing the properties in the following way:

1. There are some basic properties that are automatically generated,
2. The user can write context-dependent properties using PV tool variables, which will generate the final UPPAAL properties.
3. PV tool provides annotations to store the properties
4. Within this chapter, we give some guidelines on the properties that can be checked.

6.3.1 Reachability and Deadlocks

The most basic verifications that can be executed are the reachability checks, and deadlocks.

The reachability checks mean that: every state shall be reachable, or in other words, every state shall be reached in at least one of all possible executions. If it is not, it means that there is “dead code/model”, or there is another error.

Deadlock means a state of a set of thread instances, where no further transition can be triggered.

However, deadlock check is limited within UPPAAL and means that globally there is no deadlock. It could be that there is indeed a deadlock within a subsystem (for example two threads, one waiting for each other), but because there is another subsystem which continues, there is globally no deadlock and this generic check does not detect this.

For reachability and deadlocks the properties are automatically generated. The deadlock property is:

\texttt{A[] not deadlock}

For each thread instance and for each location, there is a reachability property of the form:

\texttt{E<> (Thread instance name). (location name)}, for example:
6.3.2 Preconditions, Postconditions, Assertions

Preconditions, postconditions and assertions are statements that shall be true at a given place in the model. Moreover, we define preconditions and postconditions as subclasses of assertions:

1. Preconditions are assertions that shall be valid before some code block (typically a method),
2. Similarly, postconditions are assertions that shall be (are) guaranteed after some code block.

For this, we use the annotation `@assert` with key `expression`. Contrary to already existing annotation, this annotation does not generate any new state or location. Rather, it puts an assertion in the place where the assertion is located.

To determine the location the PV variable `$(here)` meaning the current location name, is used. To define an assertion valid “here” (i.e. in a given place of code), we write:

```
A[] $(here) -> conditionToHold
```

Let’s see an example:

```
    17  /**
    18       * @transition:
    19       */
    20       a = 0;
    21
    22  /**
    23       * @transition:
    24       * @assert expression = "A[] $(here) imply $(instance).b == 1";
    25       */
    26       b = a + 1;
```

Note that for property specification, the variables shall be prefixed by the instance name.
This generates the following output:

**Figure 6-114: Assertion: output 1**

As you can see, there is NO assertion within the model – the assertion is specified separately, in property list:

**Figure 6-115: Assertion: output 2**

### Specification of Introduced Annotations

#### Table 6-28: Annotation @assert

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@assert</td>
</tr>
<tr>
<td>Location</td>
<td>In front of any executable statement</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (can be with @thread annotations)</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be several @assert annotations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>UPPAAL property</td>
<td>Correct UPPAAL property</td>
<td>Error, it is obligatory</td>
</tr>
</tbody>
</table>

### 6.3.3 Invariants

Invariants are expressions that shall globally hold, regardless of the location. They express: either conditions that shall always be true – i.e. determining the expected correct state of the system, or conditions that shall always be false – “bad things shall never happen”.
For this, we also use the annotation `@assert`, we just don’t need to specify the location where the condition shall hold. Therefore, the format is:

\[ A[\] \text{conditionToHold} \]

### 6.3.4 Complex Temporal Logic Expressions

As we enable the developer to directly express the UPPAAL properties, and the PV tool parses only the PV variables, the developer can define even very complex and advanced CTL properties.

However, some more support from PV tool is possibly needed – this needs further investigation.
7 Complete Use Case Example

To demonstrate the applicability of the use cases, we present here an example of application that is modeled with PV tool and analyzed with UPPAAL.

As application example, we use a variation of an example application that is provided with UPPAAL tool: a train-gate example, modeling trains passing a shared gate.

Within this use case example, we use various annotations and functionalities provided by PV tool, in particular:

1. Classes, constructors, methods,
2. Loops and conditionals,
3. Message passing,
4. Libraries
5. Properties,

Whenever possible we use default values for annotations, to show that little annotation code is indeed needed.

Our approach is to:

1. First implement the example in Java using typical Java constructs, and following Java “philosophy” in writing code,
2. And then create the resulting PV tool model by hand (tool not fully implemented), but fully following the use case specifications.

Only for the purpose of this document, we introduce first the logic of the application to the readers.

This is exactly the typical usage scenario of the tool, i.e. where there is no separate software/system design – rather, there is the knowledge and parts of the design “hidden” within the source code.
7.1 Functional Overview of the Application

To demonstrate the usefulness of our approach, we validate it by using an overall example of distributed application. A train-gate example is provided together with UPPAAL tool, and we use it as a basis for our modified example.

We have modified the UPPAAL train-gate example for a few reasons:

1. The original example has a major problem: if the train does not receive the stop message from the gate within a deadline, it does go. For safety reasons it is not acceptable. We do it reverse: the train shall receive an allowance to go within a time, otherwise it stops;
2. The original model assumes zero-delay delivery of stop message and zero-delay execution of gate software. We do not take this approximation;
3. The original model is very compact. We, in contrary, follow the modeling rules, which generates a bit more states and transitions (like for appending a new location, or using an existing one in loops), but make the model more readable.
4. Finally, the original model is custom-made for a given example, and is very “tuned” having application-specific simplifications. In contrary, we apply generic translation rules.

The functional logic of the train is as follows (introduced only for documenting purposes, to explain to the reader of this document):

1. The train has 3 main states: traveling, waiting for crossing and crossing.
2. The train is initially traveling,
3. Now, in the loop:
   a. It sends a message to the gate asking for a permission to cross,
   b. If there is NO yes-answer within 10 seconds, then it stops and waits for the permission until it arrives,
   c. Then it crosses the gate,
   d. Then it notifies the gate about the departure.

The functional logic of the gate is as follows:

1. The gate has two main states: free and busy. It has also a list of waiting trains,
2. Initially, the gate is free and has no waiting trains,
3. Now, in the loop:
   a. The gate checks if there is at least one waiting train. If not, it waits until there is the request for crossing, and adds it to the waiting list,
   b. Now, the gate is in the state that there is at least one train waiting for crossing,
   c. The gate takes the first train and sends him the permission to go,
   d. The gate waits for the notification of gate departure,
   e. The notification departure shall be from the train that was permitted to go.
### 7.2 PV Tool Input – Annotated Java Code

The developer the Java code and annotates it with PV tool annotation. The Train code looks as follows:

```java
public class Train {

    /**
     * Main method, representing the train process.
     * 
     * @param args Unused.
     * @process quantity = 3;
     * @threadMain:
     */

    public static void main(String[] args) {

        /**
         * @callMethod:
         */

        Train train = new Train();

        /**
         * @call:
         */

        train.handle();

    }

    /**
     * @methodNew:
     */

    public Train() {

    }

```
public void handle() {

    /**
     * The train is initially out of the gate and not waiting.
     */

    /**
     * Notify the gate that the train is approaching.
     * @param message "qApproachNotify", senderAddress = "me",
     * receiverAddress = "0";
     */
    notifyGateApproaching();

    /**
     * When entering to the gate, the train shall receive the permission
     * within 10 seconds.
     */
    boolean isPermissionOnTime = getPermissionToCrossInTimeOut(10);

    /**
     * If the permission does not come within 10 seconds, the train stops.
     */
    if (!isPermissionOnTime) {
        /**
         * Stop train.
         */
        stopTrain();
    }

    /**
     * Wait for the permission from the gate.
     * @param message "qCrossAllowed", senderAddress = "0",
     * receiverAddress = "me";
     */
    waitForPermissionToCross();

    /**
     * Now the train has the permission from the gate (it either received it
     * within the time, or it stopped and received it later), so
     * it crosses.
     */
    cross();

    /**
     * After crossing, the train notifies the gate.
     */
    notifyDepartureToGate();
}
The Gate code looks as follows:

```java
* Gate:
/*
public class Gate {

/**
 * Main method, representing the train.
 * @param args Unused.
 * @param process;
 * @threadMain isLooping = "no";
 */
public static void main(String[] args) {
    /**
     * @callNew:
     */
    Gate gats = new Gate();

    /**
     * @call;
     */
    gats.handle();
}

/**
 * @methodNew;
 */
public Gate() {
}
```

Figure 7-3: Gate, part 1
```java
    public void handle() {
        /* Method */
        
        // declaration
        List<Integer> waitingTrains = new ArrayList();
        int comingTrainId;
        int crossingTrainId;
        int leavingTrainId;
        
        while (true) {
            // Now, there are no trains crossing the gate.
            // Check if there is any train waiting for crossing.
            // If no train is waiting, then wait for it.
            if (waitingTrains.size() == 0) {
                // Wait for the 1st train.
                // receive message = "qApproachNotify",
                // receiverAddress = "me";
                comingTrainId = waitForRequestToCross();
                
                // Add train to the list at the end.
                waitingTrains.add(comingTrainId);
            }
            
            // There is still no train on the gate, but now we have at least
            // waiting train. Let's get the first one.
            // transition update = "waitingTrainsGet(0)";
            crossingTrainId = waitingTrains.get(0);
            
            // Now, we allow the first train to cross.
            // send message = "qCrossAllow", senderAddress = "me",
            // receiverAddress = "crossingTrainId";
            sendPermissionToCross(crossingTrainId);
        }
    }
```
The first train is shown. The others are identical, because RMI is not used.
Figure 7-6: Train: output model

- $g_{Route}(me) = -1$
  - $1001$

- $g_{Route}(me) = 0$
  - $1002$

- $g_{ApproachNotify}$
  - $g_{Route}(me) = -1, g_{Sender} = me$
  - $1003$

- $t = 0$
  - $1004$

- $t > 10$
  - $isTimeout = true$
    - $(t \leq 10) \&\& (g_{Route}[0] == me)$
      - $g_{CrossAllow}?$
        - $isTimeout = false$

- $if(isTimeout == true)$
  - $isTimeout == true$
    - $1007$

- $(g_{Route}[0] == me, g_{CrossAllow}?$
  - $1008$

- $g_{Route}(me) = 0$
  - $1009$

- $g_{DepartureNotify}$
  - $g_{Route}(me) = -1, g_{Sender} = me$
  - $1010$
The verification properties included in this example are:

1. Deadlock check
2. Reachability check
3. One custom property.

The example used custom property is as follows:

When the gate receives the id of the leaving train, this id shall be the same as the id that was last sent with the permission to go. In other words, the train that just left shall be the one that was allowed. In CTL expression, this is:

```
Query

∀[gGate00.0011] ⊨ gGate00.leavingTrainId == gGate00.crossingTrainId
```

Figure 7-8: A formal property for train-gate
8 Report on the Implementation

The scope of the diploma thesis was the research and the specification of the PV tool, as well as proof-of-concept implementation.

The first few use cases have been implemented. For the following ones, often just the core classes and methods were created, and some simplified implementation of core logic. The implementation follows Sun’s Java coding guidelines and naming conventions for Java code and Java comments and respects best programming practices, in particular as presented in R3. The source code is fully commented with Javadoc comments. Finally, the source code is (automatically) formatted according to Sun’s Java formatting style. Version control is used for all the source code (including PV tool and use cases input and outputs).

New features of Java 5 have been used extensively, in particular type-safe Maps and Lists using generics.

The PV tool has a graphical front-end, which is a plug-in for Eclipse tool. Below are some screenshots.

Figure 8-1: PV plug-in: configuration
The generated model is automatically checked by PV tool, and the results are displayed in Eclipse IDE:

Figure 8-2: PV Plug-in: error display

Moreover, the generated model can be displayed in UPPAAL tool and analyzed by the developer. The developer can in particular simulate the model, and change it to see how it influences the behavior.

Figure 8-3: UPPAAL tool: simulation
9 Further Work

Within the diploma thesis, we presented the use cases and specification that is according to us the highest priority. Together, the specified features, when implemented, will enable already to check a range of distributed applications.

However, we see that still further work can be done, in particular in the following areas:

1. deeper analysis of object-oriented mapping,
2. other inter-process communication mechanisms (broadcasts, unreliable communication),
3. more automatism in modeling standard Java stacks (like TCP, UDP, IP),
4. realtime analysis,
5. synchronized block, in particular sync between different remote methods of the same object.
6. Inter-thread synchronization,
7. Deeper analysis of properties, especially in respect of generation of set of properties from meta-properties annotated in code.
10 Verification, Validation and Conclusion

The PV tool extracts the model and properties from Java code and validates the model against properties. The main differences of our approach with respect to the state-of-art is that the extraction of the model is developer-driven thanks to code annotations, enabling to reach high level of abstraction, selection of only model elements that are relevant for property verification and finally enabling to get an understandable, simple model that can be even simulated by the developer. We avoid state-explosion and scalability issues.

To validate specification demonstrated in this use case, we have done a concrete example of a simple distributed system, which is a modified train-gate example, provided together with UPPAAL program release.

Whereas within the use cases we were introducing the features and showing how they can be used, in the complete example we did the reverse: we took the existing model and checked if our specification works.

This came out to be very useful for the PV Tool:

1. We found an important feature missing (busy waiting on message reception with a timeout) – we decided not to consider time/realtime aspect with the use cases, but this basic functionality has been appended as it is important,

2. We detected some minor errors, inconsistencies and ambiguities within the use cases, and correspondingly updated them,

Concerning the example itself, we demonstrated that the approach is working and can efficiently find errors in distributed application.

Firstly, through the verification of properties, we have found that they are not respected.

Secondly, we have found the around five “stupid” errors in code (examples: wrong if-condition forgotten removal of element from array resulting in deadlock). These errors were found just through the model simulation. The simulation has been done randomly, and by manual simulation (by selection of transitions by developer).

We worked on bug finding until all the properties were respected.

It came out that the graphical interface and user-driven simulation of timed automata together with random simulation is a very efficient way for finding bugs.
## 11 References

### 11.1 Documents

<table>
<thead>
<tr>
<th>Ref</th>
<th>Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>R5</td>
<td>Bastian Schlich: “Verification of Discrete Control Systems”, RWTH Aachen</td>
</tr>
<tr>
<td>R7</td>
<td>Javadoc specification, <a href="http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/Javadoc.html#tag">http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/Javadoc.html#tag</a>, 2006</td>
</tr>
<tr>
<td>R8</td>
<td>Joshua Bloch: “Effective Java / Programming Language Guide”</td>
</tr>
</tbody>
</table>
### 11.2 Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>LTL</td>
<td>Linear Temporal Logic</td>
</tr>
<tr>
<td>CTL</td>
<td>Computational Temporal</td>
</tr>
<tr>
<td>DoDAF</td>
<td>Department of Defense Architecture Framework</td>
</tr>
<tr>
<td>JML</td>
<td>Java Modeling Language</td>
</tr>
<tr>
<td>SIL</td>
<td>Safety Integrity Level</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
</tbody>
</table>
This chapter specifies all PV tool annotations, which have been introduced in the use cases. For each annotation, it specifies all needed details, including meaning, default values and meaning and the corresponding output in UPPAAL.

12.1 General Annotation Syntax

The annotations used by PV tool can be attachment to any Java statement: not only to a method, class or interface, but to any line in the body of any method. The annotations respect Javadoc syntax and they are composed of two parts:

1. One description sentence (line 12 in the example),
2. One or more custom Javadoc tags with its value (line 16 in the example), where the value is an JML expression.

The overall example looks like follows:

Code:
```java
/**
 * Description sentence.
 * @param test Test parameter.
 * @return test result.
 * @customTag0 JMLExpression0
 * @customTag0 JMLExpression1
 * @customTag1 JMLExpression2
 */
int testMethod(int test) {
```

Some tags may appear only once per Java statement (in example it is @customTag0), some others can appear several times per Java statement. The JML expression is a of key=property pairs separated by comma. At the end there is a semicolon. The strings are within quotes and the arrays are within parenthesis. Boolean values are either true or false, without quotes. An example can look as follows:
12.2 PV Tool Annotations

There are different annotation types, and for each annotation type there is a different custom Javadoc tag. The following table lists all currently available custom tags and the keys/values that are used within their context:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@process</td>
</tr>
<tr>
<td>Location</td>
<td>In front of main() method</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Possible with @threadMain and @rmiRoute</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur once per Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Short name of the process</td>
<td>Same as Java class naming</td>
<td>Class name to which main() belongs</td>
</tr>
<tr>
<td>quantity</td>
<td>Quantity of process instances</td>
<td>Integer &gt;= 1</td>
<td>1</td>
</tr>
</tbody>
</table>

```java
.../**
... * Description sentence.
... * @param test Test parameter.
... * @return test result.
... * @customTag key1=1, key2="test", key3={1, 2}, key4=0;
... */
... int testMethod2(int test) {
```
### Table 12-2: Annotation @threadMain

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@threadMain</td>
</tr>
<tr>
<td>Location</td>
<td>In front of main() method</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Possible with @process and @rmiRoute</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur once per Java statement</td>
</tr>
<tr>
<td>Key</td>
<td>Meaning</td>
</tr>
<tr>
<td>isLooping</td>
<td>Whether the end state shall be connected to start state</td>
</tr>
</tbody>
</table>

### Table 12-3: Annotation @threadRunnable

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@threadRunnable</td>
</tr>
<tr>
<td>Location</td>
<td>In front of run() method of class implementing Runnable interface</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No, @rmiRoute is allowed</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur once per Java statement</td>
</tr>
<tr>
<td>Key</td>
<td>Meaning</td>
</tr>
<tr>
<td>name</td>
<td>Short name of the thread</td>
</tr>
<tr>
<td>quantity</td>
<td>Quantity of thread instances within one process instance</td>
</tr>
<tr>
<td>isLooping</td>
<td>Whether the end state shall be connected to start state</td>
</tr>
</tbody>
</table>
### Table 12-4: Annotation @threadRemote

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@threadRemote</td>
</tr>
<tr>
<td>Location</td>
<td>In front of the class implementing RemoteUnicastObject interface</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No, @rmiRoute is allowed</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur once per Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>interfaceName</td>
<td>Shortened name of the Remote interface, which is used by clients (in lookup)</td>
<td>Same as Java interface naming</td>
<td>The remote interface name</td>
</tr>
<tr>
<td>quantityOfServantsPerMethod</td>
<td>Quantity of thread instances (servants) per each remote method and per one remote object</td>
<td>Integer (\geq 1)</td>
<td>1</td>
</tr>
<tr>
<td>quantityOfObjects</td>
<td>Quantity of objects implementing the remote interface within one process instance</td>
<td>Integer (\geq 1)</td>
<td>1</td>
</tr>
<tr>
<td>isLooping</td>
<td>Whether the end state shall be connected to start state</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>
## Table 12-5: Annotation `@transition`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@transition</code></td>
</tr>
<tr>
<td>Location</td>
<td>Any statement within a method</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>None</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur several times for one Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>select</code></td>
<td>Selection of random value</td>
<td>Conformant UPPAAL Select expression</td>
<td>Empty string</td>
</tr>
<tr>
<td><code>guard</code></td>
<td>Unused, undefined</td>
<td>Unused, undefined</td>
<td>Empty string</td>
</tr>
<tr>
<td><code>update</code></td>
<td>Quantity of thread instances within one process instance</td>
<td>Integer ≥ 1</td>
<td>Java expression to which the annotation is attached (regardless of the existence of other annotations for the Java statement)</td>
</tr>
<tr>
<td><code>sync</code></td>
<td>Synchronization (reception/send of a signal) from/to another template</td>
<td>Channel access expression</td>
<td>Empty string</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Annotation</td>
<td>@declaration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Location</td>
<td>Any statement within a method</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exclusivity</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplicity</td>
<td>Can occur several times for one Java statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Key</strong></td>
<td><strong>Meaning</strong></td>
<td><strong>Value constraints</strong></td>
<td><strong>Assumed value if not specified</strong></td>
</tr>
<tr>
<td>expression</td>
<td>Declaration of variable</td>
<td>Conformant UPPAAL declaration.</td>
<td>Java expression to which the annotation is attached (regardless of the existence of other annotations for the Java statement)</td>
</tr>
<tr>
<td>isGlobal</td>
<td>Wheter it is declared within the template, or globally</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
Table 12-7: Annotation @loopFor

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@loopFor</td>
</tr>
</tbody>
</table>

Location In front of for loop

Exclusivity Remaining annotations of the statement can be @transition and @declaration.

Multiplicity There can be only one @loopFor annotation per Java statement

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>Initialization expression in the loop</td>
<td>Conformant UPPAAL update expression.</td>
<td>Initialization statement within Java for loop</td>
</tr>
<tr>
<td>guard</td>
<td>Guard condition for the loop</td>
<td>Conformant UPPAAL guard expression.</td>
<td>Guard condition within Java loop</td>
</tr>
<tr>
<td>Update</td>
<td>Update (counter incrementation) expression in the loop</td>
<td>Conformant UPPAAL update expression.</td>
<td>Guard condition within Java loop</td>
</tr>
</tbody>
</table>

Table 12-8: Annotations @loopWhile, @loopDoWhile

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@loopWhile, @loopDoWhile</td>
</tr>
</tbody>
</table>

Location In both cases, in front of while() statement

Exclusivity Remaining annotations of the statement can be @transition and @declaration.

Multiplicity There can be only one @loopWhile annotation per Java statement

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>guard</td>
<td>Guard condition for the loop</td>
<td>Conformant UPPAAL guard expression.</td>
<td>Guard condition within Java loop</td>
</tr>
</tbody>
</table>
### Table 12-9: Annotations @loopDoWhile

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@loopDoWhile</td>
</tr>
<tr>
<td>Location</td>
<td>In front of <code>while()</code> statement.</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Remaining annotations of the statement can be <code>@transition</code> and <code>@declaration</code>.</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one @loopWhile annotation per Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>guard</td>
<td>Guard condition for the loop</td>
<td>Conformant UPPAAL guard expression.</td>
<td>Guard condition within Java loop</td>
</tr>
</tbody>
</table>

### Table 12-10: Annotations @if, @ifElseIf, @ifElse

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@if, @ifElseIf, @ifElse</td>
</tr>
<tr>
<td>Location</td>
<td>In front of statements: if, else if and else respectively</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Remaining annotations of the statement can be <code>@transition</code> and <code>@declaration</code>.</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one @if, @ifElseIf, @ifElse annotation per Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>guard</td>
<td>Guard condition for the loop</td>
<td>Conformant UPPAAL guard expression.</td>
<td>Guard condition within Java loop</td>
</tr>
</tbody>
</table>
Table 12-11: Annotations `@loopSwitch`, `@loopSwitchCase`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@loopSwitch</code>, <code>@loopSwitchCase</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of statements: <code>if</code>, <code>else if</code> and <code>else</code> respectively</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Remaining annotations of the statement can be <code>@transition</code> and <code>@declaration</code></td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one <code>@if</code>, <code>@elseif</code>, <code>@else</code> annotation per Java statement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>guard</code></td>
<td>Guard condition for the loop</td>
<td>Conformant UPPAAL guard expression.</td>
<td>Guard condition within Java loop</td>
</tr>
</tbody>
</table>
### Table 12-12: Annotation @class

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@class</td>
</tr>
<tr>
<td>Location</td>
<td>In front of class declaration</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Yes (i.e. no other PV tool annotations allowed).</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Alternative name</td>
<td>Same as Java class naming</td>
<td>Class name</td>
</tr>
<tr>
<td>isGlobal</td>
<td>Whether the output typedefs struct shall be global or local to thread</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

### Table 12-13: Annotation @classImport

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@classImport</td>
</tr>
<tr>
<td>Location</td>
<td>In front of class import statement.</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (for example, a @declaration is used for static attributes)</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Alternative name</td>
<td>Same as Java class naming</td>
<td>Class name</td>
</tr>
</tbody>
</table>
| attribute
dclarations | List of primitive attributes with data types          | List in the form “Type value”, like {“int x”, “int y”} | Empty string                  |
Table 12-14: Annotation @attribute

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@attribute</td>
</tr>
<tr>
<td>Location</td>
<td>In front of an attribute within a class</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Yes (i.e. no other PV tool annotations allowed).</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
</table>
| Declaration    | Declaration of a primitive attribute                  | Conformant UPPAAL declaration. | Java expression to which the annotation is attached, but without the following strings (they are cut out by PV tool):  
|                |               |                            | • public/private/protected  
|                |               |                            | • static                     
|                |               |                            | • transient                  
|                |               |                            | • synchronized               |
Table 12-15: Annotation @method

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@method</td>
</tr>
<tr>
<td>Location</td>
<td>In front of an method within a class</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Yes (i.e. no other PV tool annotations allowed).</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>List of parameters with data types</td>
<td>Like for Java signature. Without parenthesis.</td>
<td>The same as Java parameters signature For example: “int x, int y”</td>
</tr>
</tbody>
</table>

Table 12-16: Annotation @return

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@return</td>
</tr>
<tr>
<td>Location</td>
<td>In front of return expression.</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>Yes (i.e. no other PV tool annotations allowed).</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Expression passed to return, copied on the return value of the caller.</td>
<td>Like for Java expression</td>
<td>The value of the return expression (without return keyword).</td>
</tr>
</tbody>
</table>

Table 12-17: Annotation @methodNew

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@methodNew</td>
</tr>
</tbody>
</table>
| Location | In front of a constructor.  
|          | Shall not be used in front of object factory. |
| Exclusivity | Yes (i.e. no other PV tool annotations allowed). |
|Multiplicity | There can be only one annotation per Java class declaration. |

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>List of parameters with data types</td>
<td>Like for Java signature.</td>
<td>The same as Java parameters signature</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Without parenthesis.</td>
<td>For example: “int x, int y”</td>
</tr>
<tr>
<td>isGlobal</td>
<td>Whether the output typedefs struct shall be</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>global or local to thread</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 12-18: Annotation @call

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@call</td>
</tr>
<tr>
<td>Location</td>
<td>In front of invocation/call of a method</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (before and after there can be transitions, declarations).</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>Expression passed to return, copied on the return value of the caller.</td>
<td>Like for Java expression</td>
<td>Java signature, For example “int x, int y”</td>
</tr>
<tr>
<td>return</td>
<td>Variable name (without data type) receiving the return value of the method call</td>
<td>Like for variable/attribute naming.</td>
<td>Java variable name (without data type)</td>
</tr>
</tbody>
</table>
### Table 12-19: Annotation `@callNew`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@callNew</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of invocation/call of a constructor</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (before and after there can be transitions, declarations.)</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
<tr>
<td>Key</td>
<td><strong>Meaning</strong></td>
</tr>
<tr>
<td>parameters</td>
<td>Expression passed to return, copied on the return value of the caller.</td>
</tr>
<tr>
<td></td>
<td>Like for Java expression</td>
</tr>
<tr>
<td></td>
<td>Assumed value if not specified</td>
</tr>
<tr>
<td></td>
<td>Java signature, For example “int x, int y”</td>
</tr>
<tr>
<td>expression</td>
<td>Class name and initial object reference</td>
</tr>
<tr>
<td></td>
<td>Naming like for classes and object references</td>
</tr>
<tr>
<td></td>
<td>Assumed value if not specified</td>
</tr>
<tr>
<td></td>
<td>Java expression</td>
</tr>
<tr>
<td></td>
<td>For example: <code>Circle c = new Circle()</code></td>
</tr>
<tr>
<td></td>
<td>Has default expression <code>Circle c</code></td>
</tr>
</tbody>
</table>

### Table 12-20: Annotation `@rmiLookup`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@rmiLookup</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of <code>Naming.lookup()</code></td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (before and after there can be transitions, declarations.)</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be only one annotation per Java class declaration.</td>
</tr>
<tr>
<td>Key</td>
<td><strong>Meaning</strong></td>
</tr>
<tr>
<td>expression</td>
<td>Class name and initial object reference</td>
</tr>
<tr>
<td></td>
<td>Naming like for classes and object references</td>
</tr>
<tr>
<td></td>
<td>Assumed value if not specified</td>
</tr>
<tr>
<td></td>
<td>Java expression</td>
</tr>
<tr>
<td></td>
<td>For example: <code>Circle c = Naming.lookup(s)</code></td>
</tr>
<tr>
<td></td>
<td>Has default expression <code>Circle c</code></td>
</tr>
</tbody>
</table>
### Table 12-21: Annotation `@rmiRoute`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@rmiRoute</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of thread declaration</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (can be with <code>@thread</code> annotations)</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be several <code>@rmiRoute</code> annotations</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value constraints</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Interface name for which the route applies</td>
<td>Existing annotated name</td>
<td>Error, it is obligatory</td>
</tr>
<tr>
<td>clientProcess</td>
<td>Id of the client process</td>
<td>Index(es) within the range, separated by comma</td>
<td>Error, it is obligatory</td>
</tr>
<tr>
<td>clientThread</td>
<td>Id of client thread, if client is a runnable thread</td>
<td>Index(es) within the range, separated by comma</td>
<td>Obligatory for Runnable threads (error if not specified).</td>
</tr>
<tr>
<td>clientObject</td>
<td>Id of client object, if client is a remote thread</td>
<td>Index(es) within the range, separated by comma</td>
<td>Obligatory for Remote threads (error if not specified).</td>
</tr>
<tr>
<td>serverProcess</td>
<td>Id of the server process</td>
<td>Index within the range</td>
<td>Error, it is obligatory</td>
</tr>
<tr>
<td>serverObject</td>
<td>Id of the server object</td>
<td>Index within the range</td>
<td>Error, it is obligatory</td>
</tr>
</tbody>
</table>

### Table 12-22: Annotation `@send`

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td><code>@send</code></td>
</tr>
<tr>
<td>Location</td>
<td>In front of any statement that sends data to remote thread</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
<th>Value</th>
<th>Assumed value if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>message</strong></td>
<td>For message-addressed communication, the message id</td>
<td>Shall be the name of existing channel,</td>
<td>Value $g\text{Network} \rightarrow$ assumes using one common network and source-destination addressing</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
<td>---------------------------------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>sender</strong></td>
<td>Sender address</td>
<td>An integer</td>
<td>Error, obligatory</td>
</tr>
<tr>
<td><strong>receiver</strong></td>
<td>Receiver address</td>
<td>An integer</td>
<td>Error, obligatory</td>
</tr>
<tr>
<td><strong>payload</strong></td>
<td>Expression copying local variables on global ones</td>
<td>UPPAAL value copy expression</td>
<td>Empty string – assumed no application data passing. Example: $g\text{Payload} = \text{payload}$</td>
</tr>
</tbody>
</table>
### Table 12-23: Annotation @receive

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@receive</td>
</tr>
<tr>
<td>Location</td>
<td>In front of any statement that receives data to Remote thread</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>No</td>
</tr>
<tr>
<td><strong>Key</strong></td>
<td><strong>Meaning</strong></td>
</tr>
<tr>
<td>message</td>
<td>For message-addressed communication, the message id</td>
</tr>
<tr>
<td>sender</td>
<td>Sender address</td>
</tr>
<tr>
<td>receiver</td>
<td>Receiver address</td>
</tr>
<tr>
<td>payload</td>
<td>Expression copying global variables on local ones, to represent data transfer</td>
</tr>
<tr>
<td>timeout</td>
<td>Time that is waited for the reception</td>
</tr>
<tr>
<td>Condition</td>
<td>Variable getting true if there was a timeout</td>
</tr>
</tbody>
</table>

### Table 12-24: Annotation @assert

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Annotation</td>
<td>@assert</td>
</tr>
<tr>
<td>Location</td>
<td>In front of any executable statement</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>No (can be with @thread annotations)</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>There can be several @assert annotations</td>
</tr>
<tr>
<td>Key</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>expression</td>
<td>UPPAAL property</td>
</tr>
</tbody>
</table>
### Table 12-25: PV Tool Variables

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Meaning</th>
<th>Example (Java → UPPAAL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(this)$</td>
<td>The initial reference name of the object in the context</td>
<td>$(this).x \rightarrow point.x$</td>
</tr>
<tr>
<td>$(here)$</td>
<td>Location name that will be in the place where the annotation <code>@assert</code> is present</td>
<td>$(here) \rightarrow Gate00.1007$</td>
</tr>
<tr>
<td>$(variable name from parameter)$</td>
<td>Variable name passed as a parameter of a method. This will be replaced by the original reference name. This is caused by different naming of objects by methods and method callers.</td>
<td>$(point).x \rightarrow destination.x$</td>
</tr>
<tr>
<td>$(process)$</td>
<td>Process name</td>
<td>See below</td>
</tr>
<tr>
<td>$(processId)$</td>
<td>Process id</td>
<td>$(process)$(processId) \rightarrow Atm00 (main thread of Atm)</td>
</tr>
<tr>
<td>$(thread)$</td>
<td>Thread name</td>
<td>See below</td>
</tr>
<tr>
<td>$(threadId)$</td>
<td>Thread id</td>
<td>See below</td>
</tr>
<tr>
<td>$(method)$</td>
<td>Method name, for Remote methods</td>
<td>See below</td>
</tr>
</tbody>
</table>
| $(template)$  | Full template name | $(template)$ is equivalent to:  
|               |           | $(process)$(processId) – for main threads  
|               |           | $(process)$(processId)$(thread) – for Runnable threads  
|               |           | $(process)$(processId)$(objectId)$(method) – for runtime threads |
| $(instance)$  | Name of the instance. | $(instance)$ is equivalent to:  
|               |           | $(template)$(threadId)  
|               |           | $(template) \rightarrow Atm00Com  
|               |           | $(instance) \rightarrow Atm00Com00 |