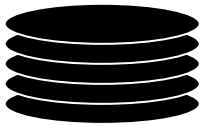
Towards D2.2 and WP2 next steps

Davide Di Ruscio – University of L'Aquila

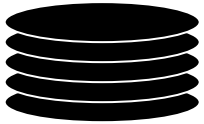
Outline

- Introduction
- Instantiation of the metamodel on a wide-used GNU/Linux distribution
  - Model Injection
  - Supporting tools
- Next steps
  - Model-based framework for managing the complexity and the state of the GNU/Linux instantiation
- Conclusions

Introduction

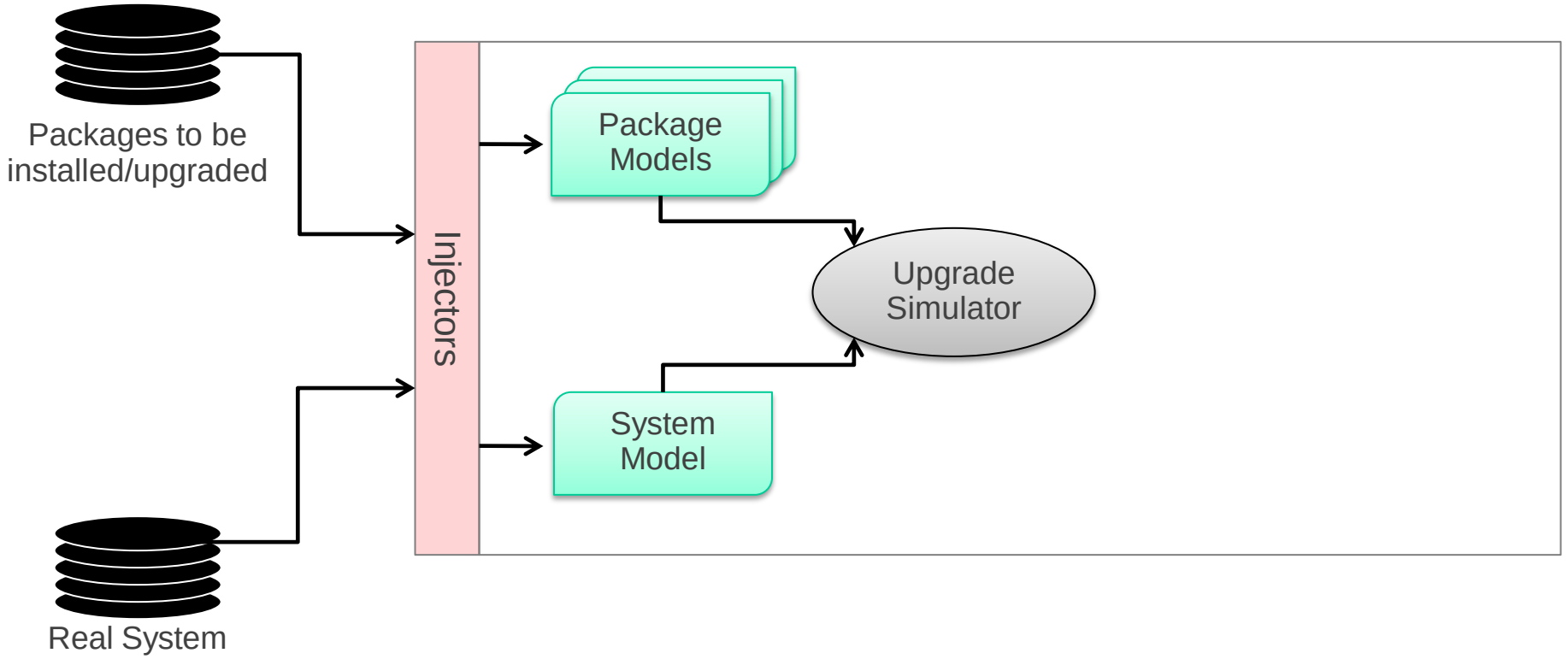


Packages to be installed/upgraded

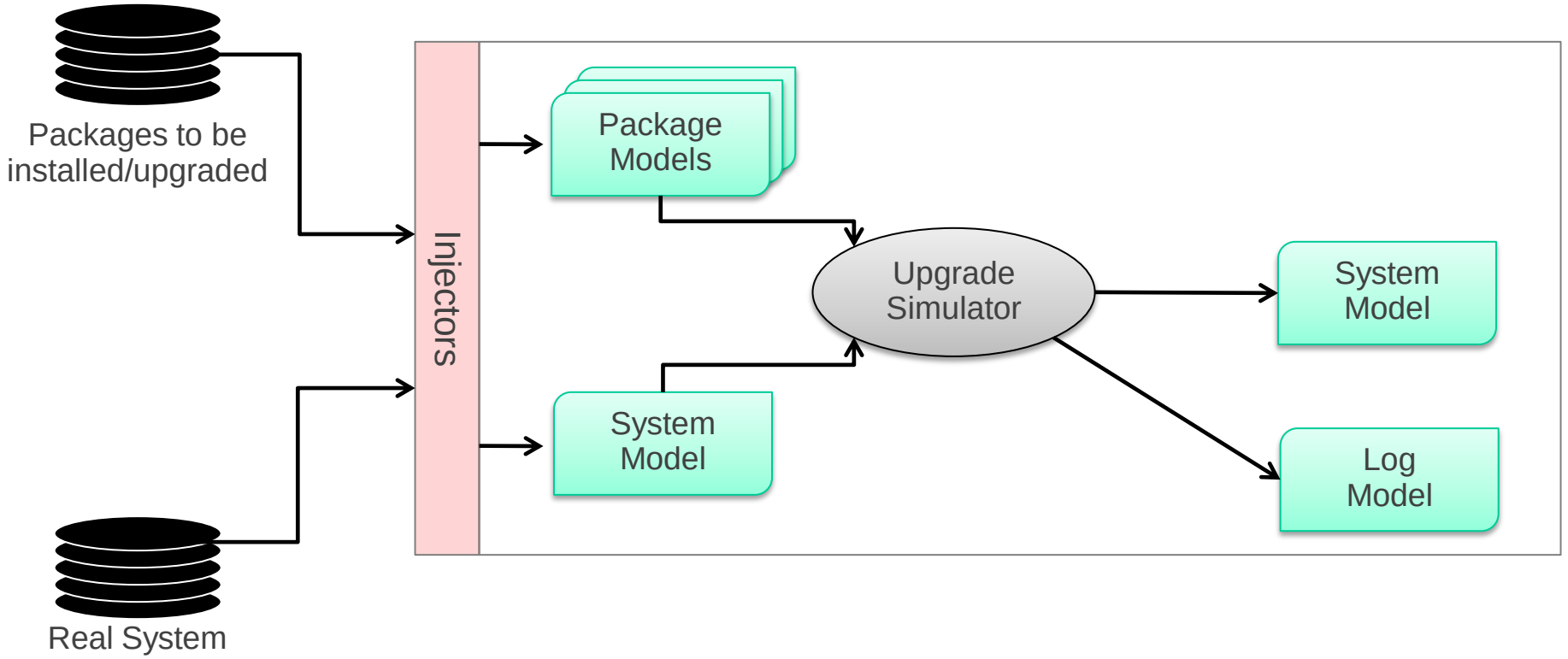


Real System

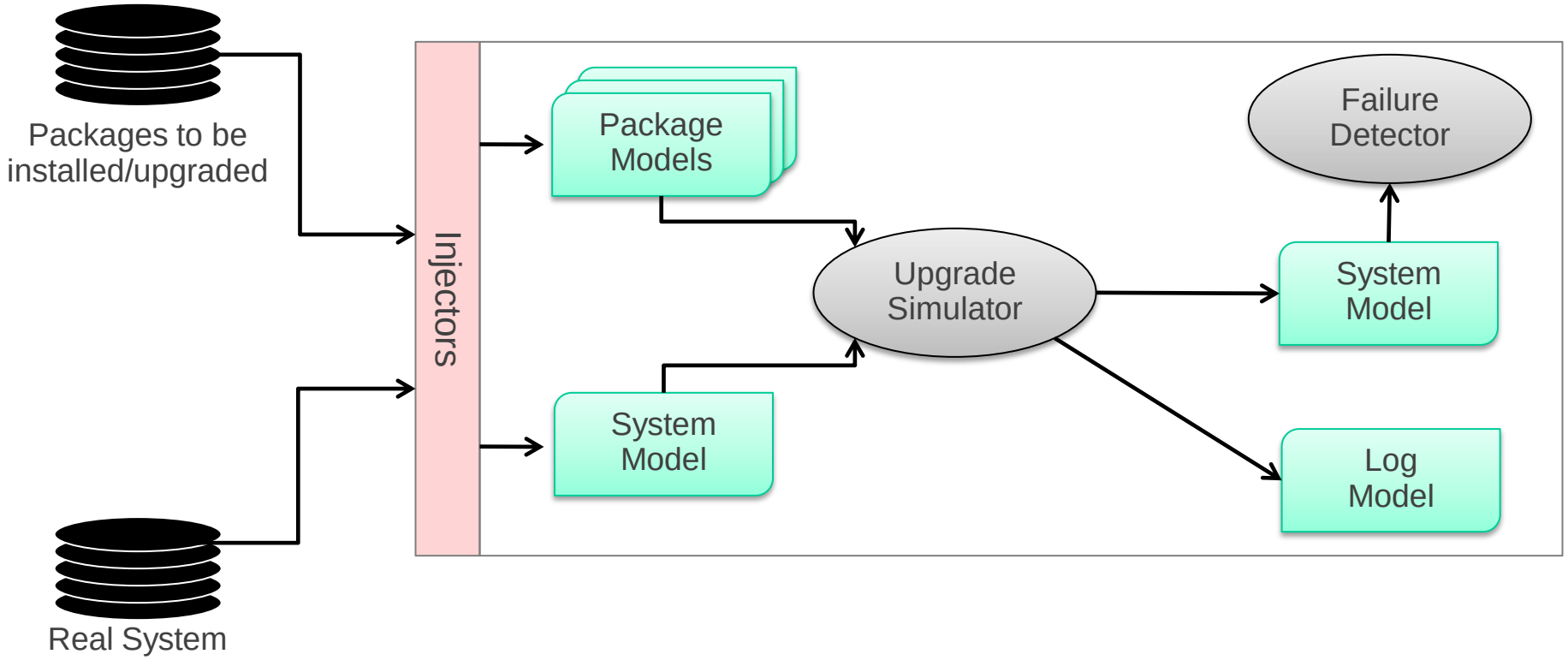
Introduction



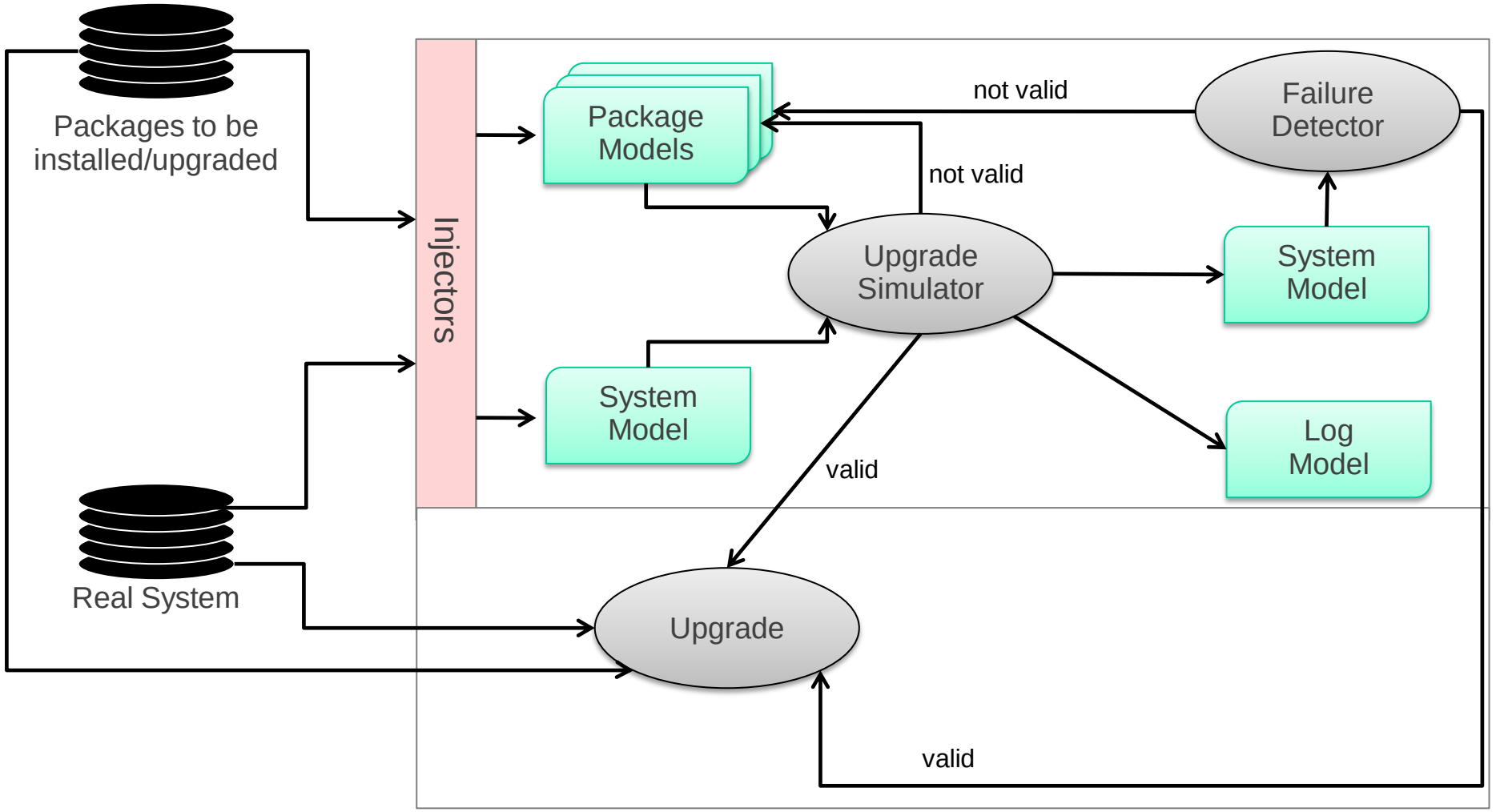
Introduction



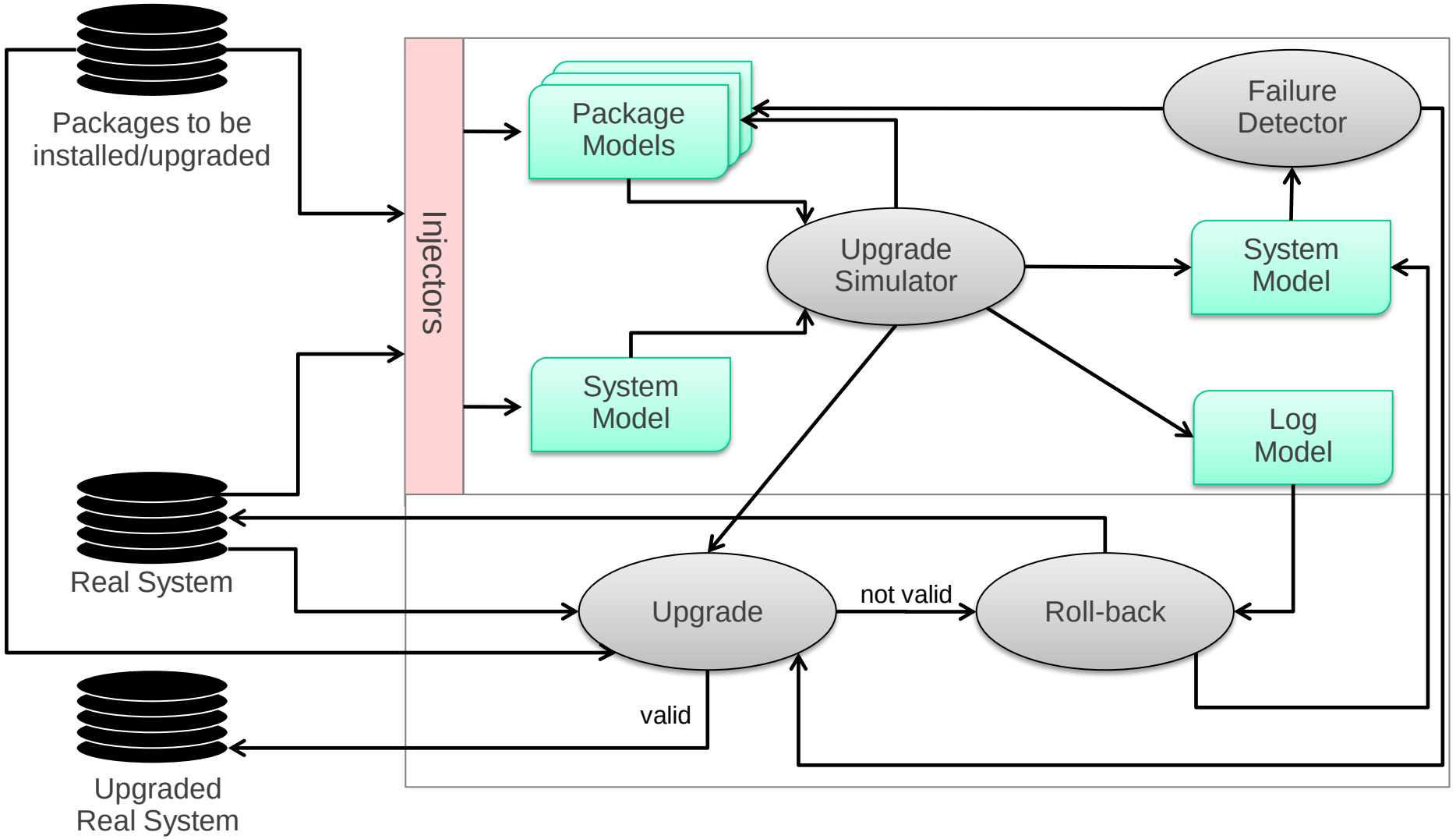
Introduction



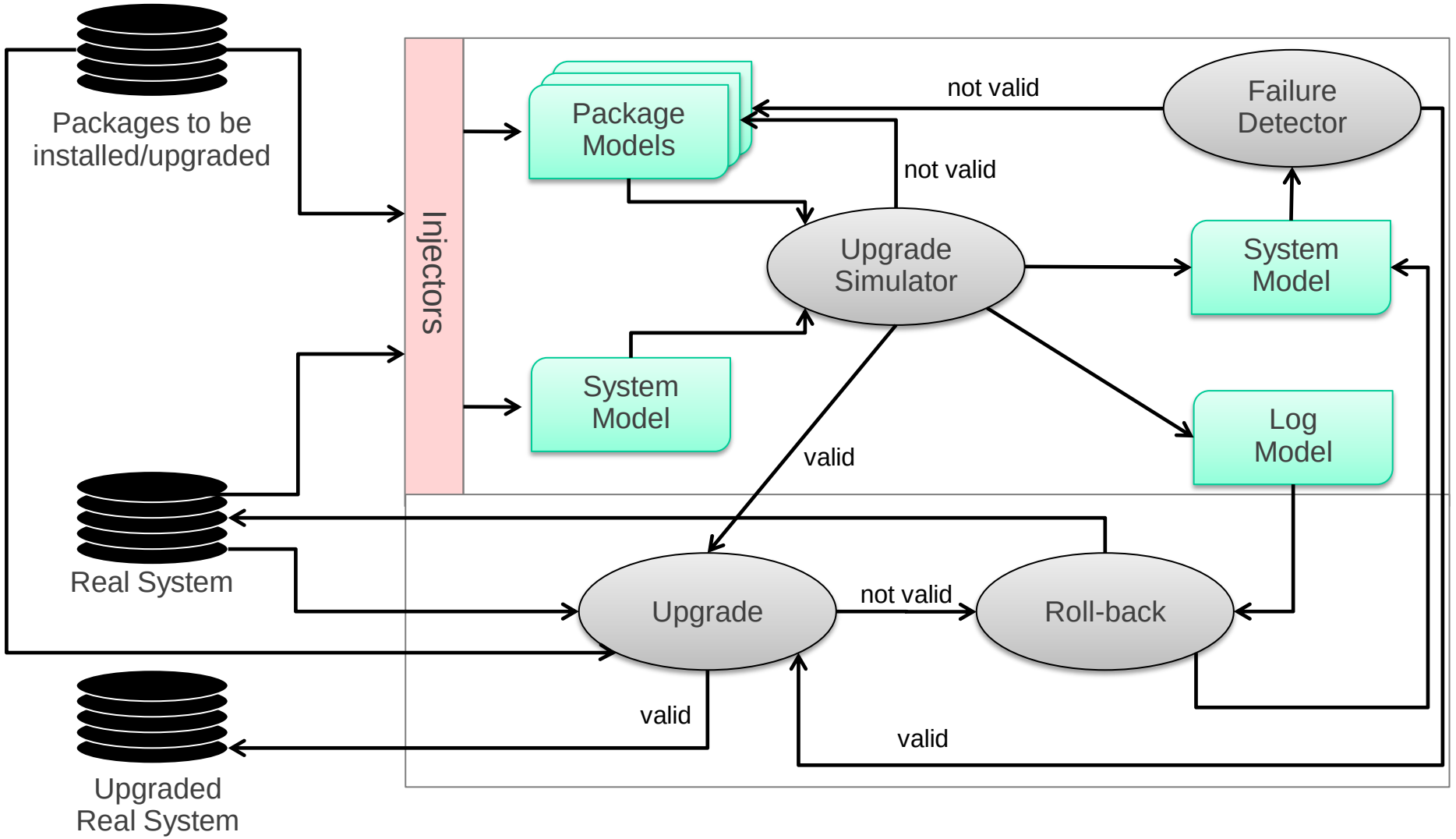
Introduction



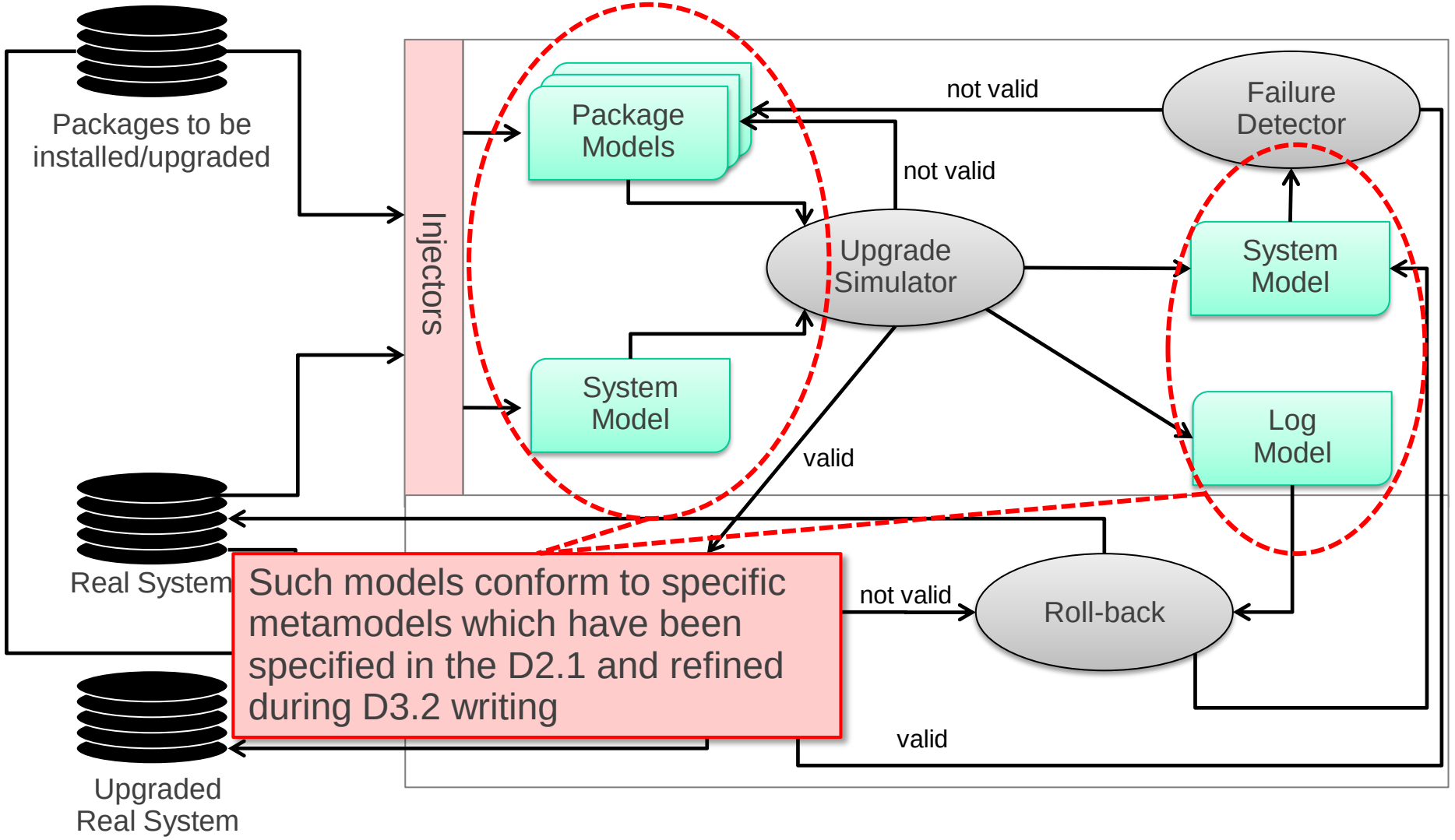
Introduction



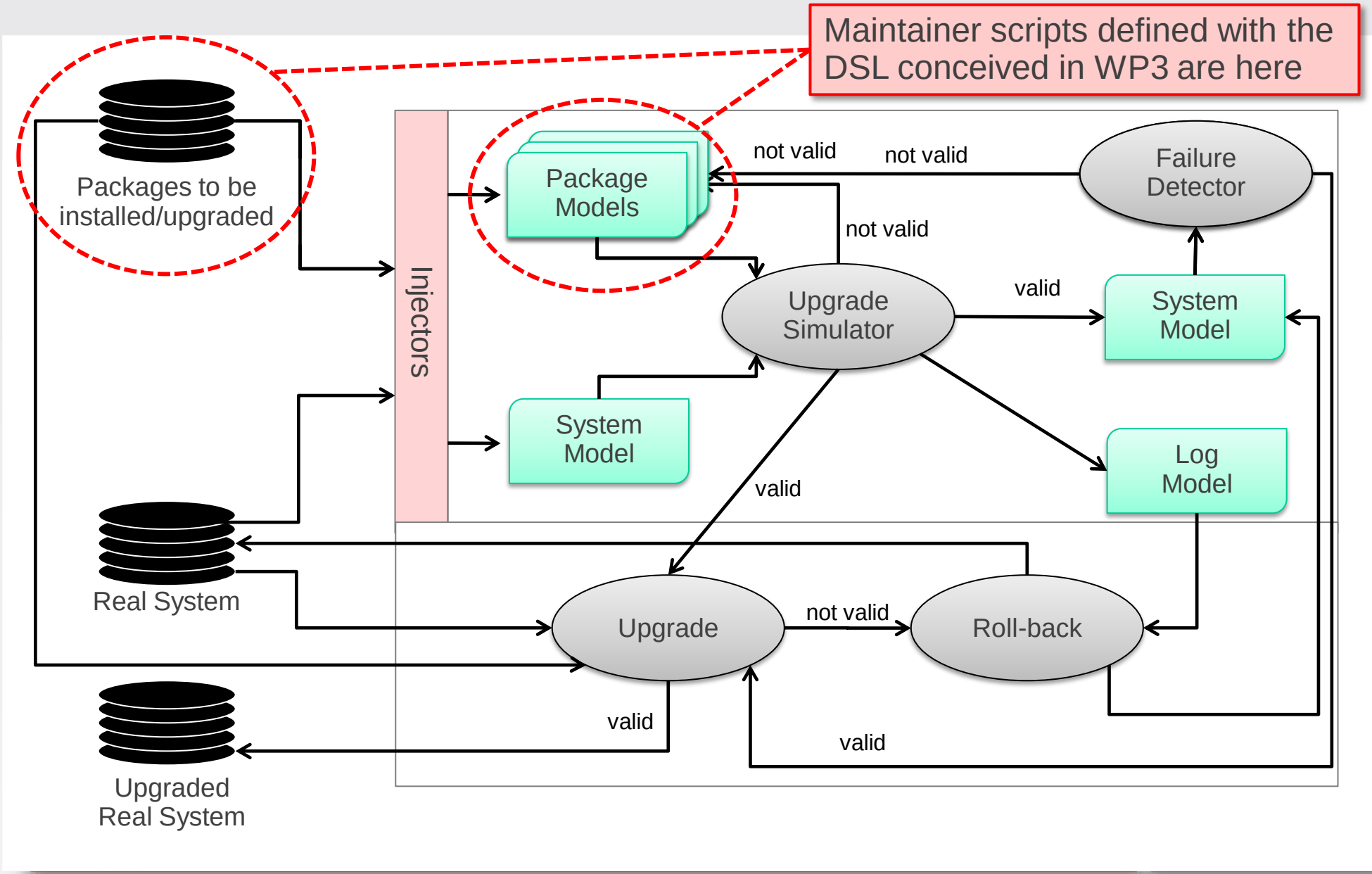
Introduction



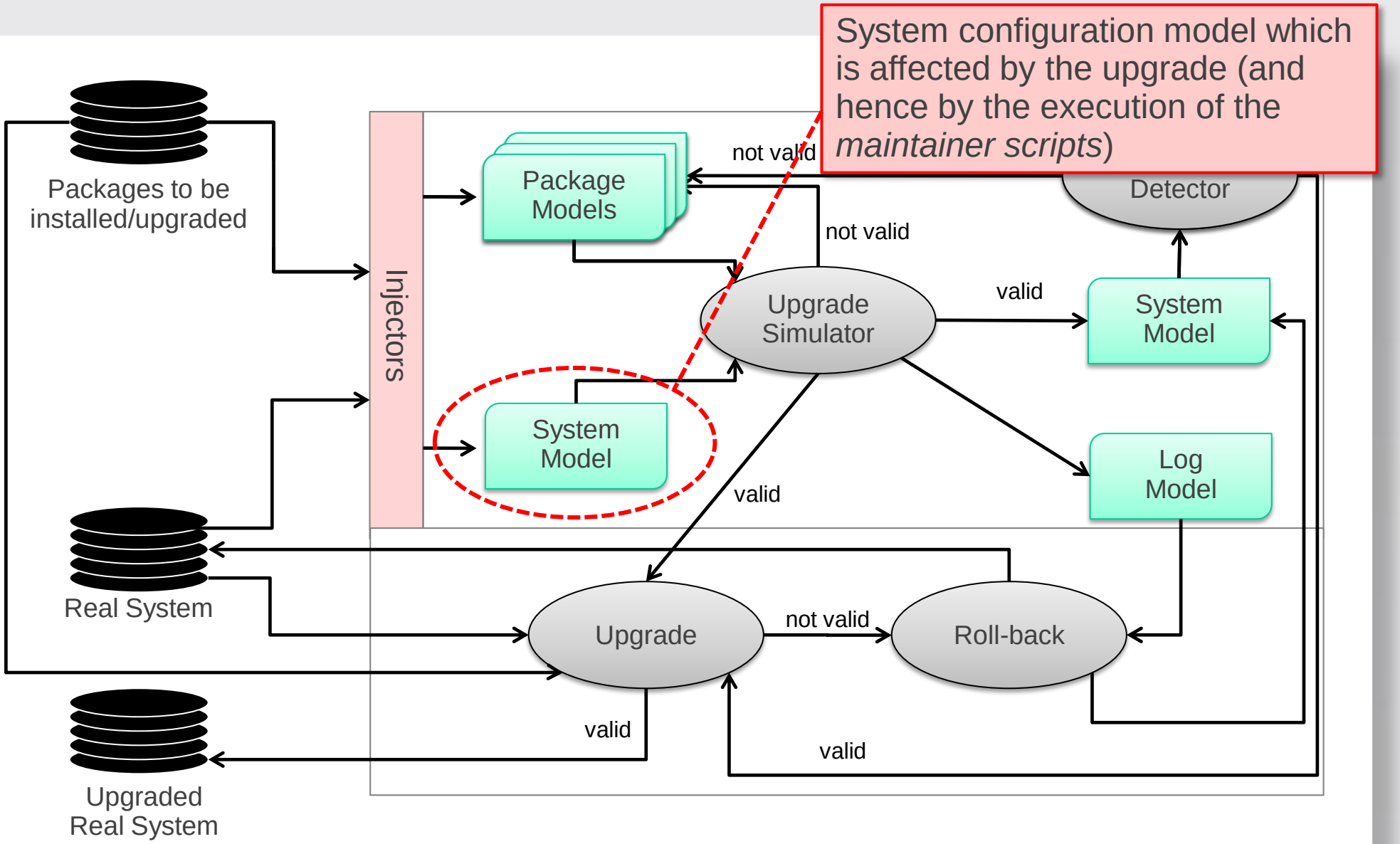
Introduction



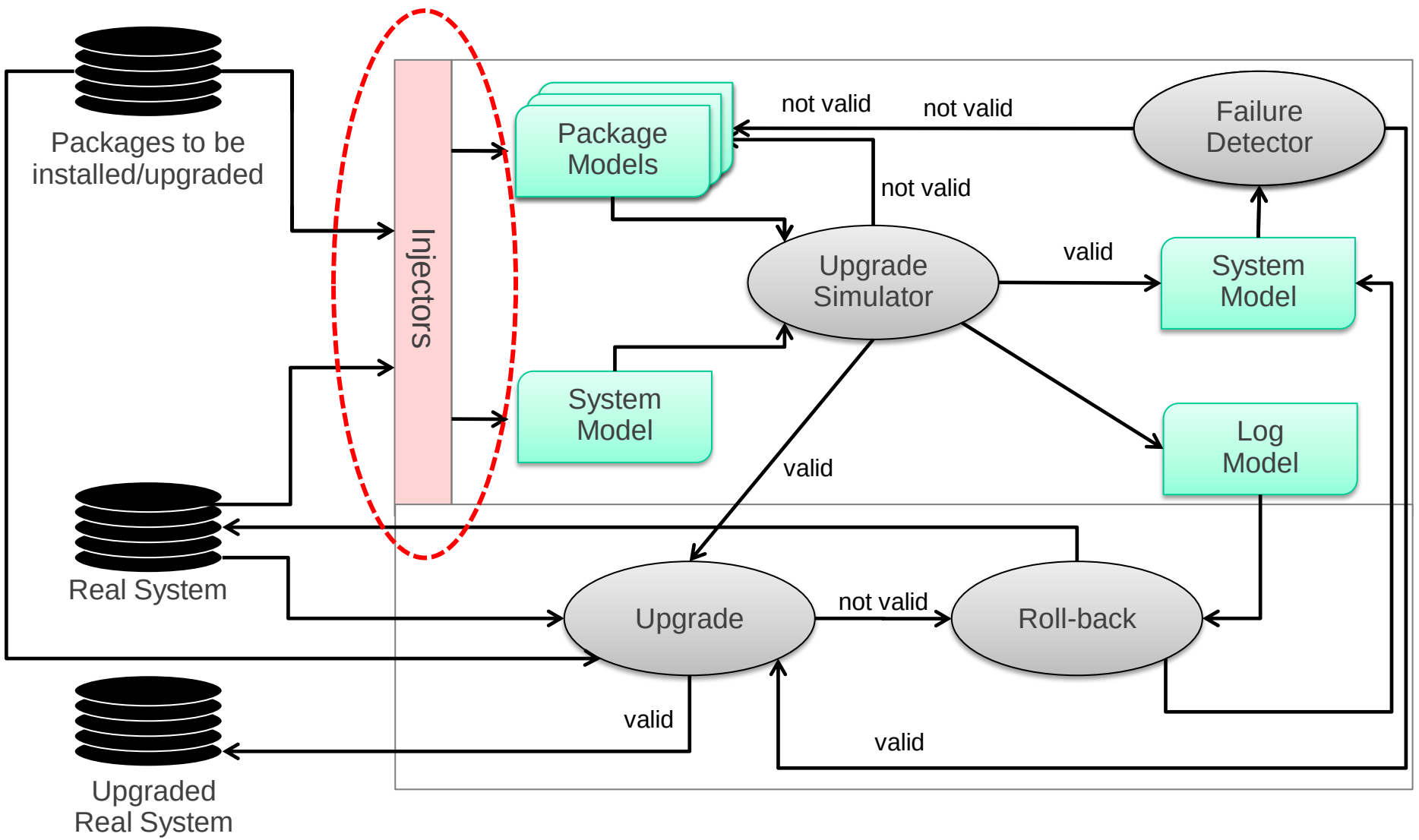
Introduction



Introduction



Introduction



Possible scenarios

» The model driven approach proposed by WP2 can be adopted in two different scenarios

» The model driven approach

1. is natively used by distributions
2. is used on systems already existing and running

Scenario 1

- There is an installed core of a Linux distribution which has a corresponding configuration model which is modified and extended during the installation of packages

- Maintainer scripts are already written by means of the DSL

Scenario 2

» Existing and running systems have to be specified in terms of models which describe system configurations

» Packages have to be represented in terms of models and their maintainer scripts have to be specified by using the DSL

Towards D2.2

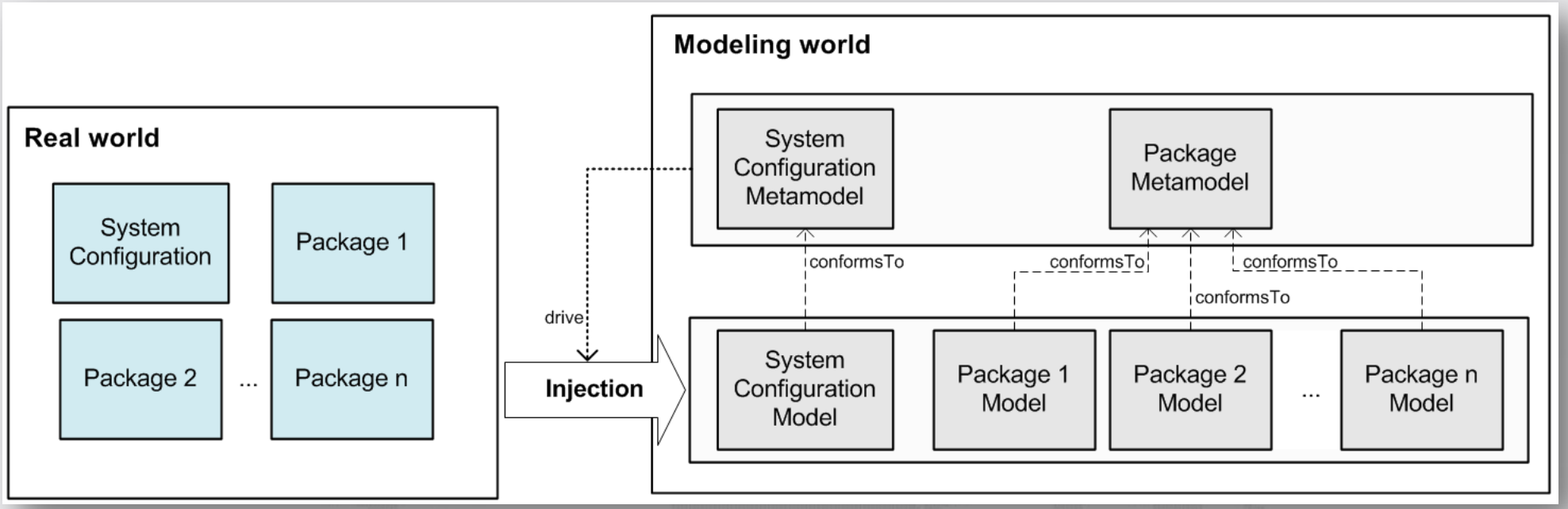
Title of D2.2: Instantiation of the metamodel on a wide-used GNU/Linux distribution

Due at: T0+24

- The deliverable has to describe techniques and tools which have been adopted and conceived to deal with the second scenario
- Starting from an existing Linux installation, corresponding models conforming to the MANCOOSI metamodel (presented in D2.1 and refined during the D3.2 writing) have to be produced

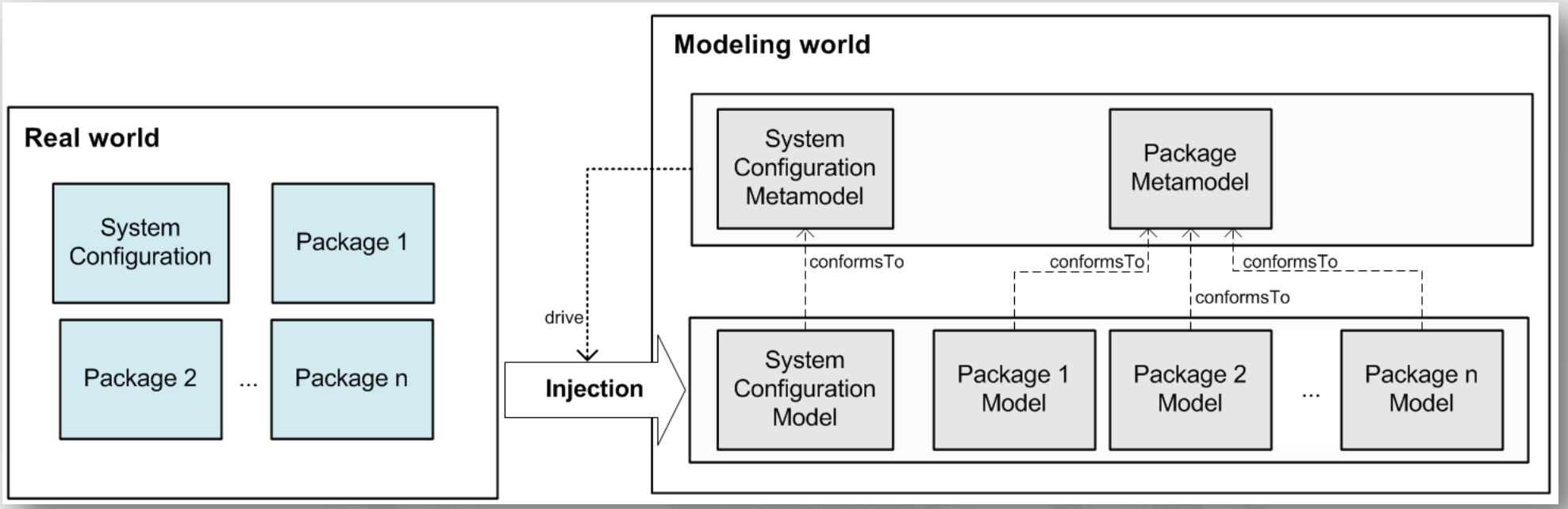
  => *Model Injection*

Model Injection



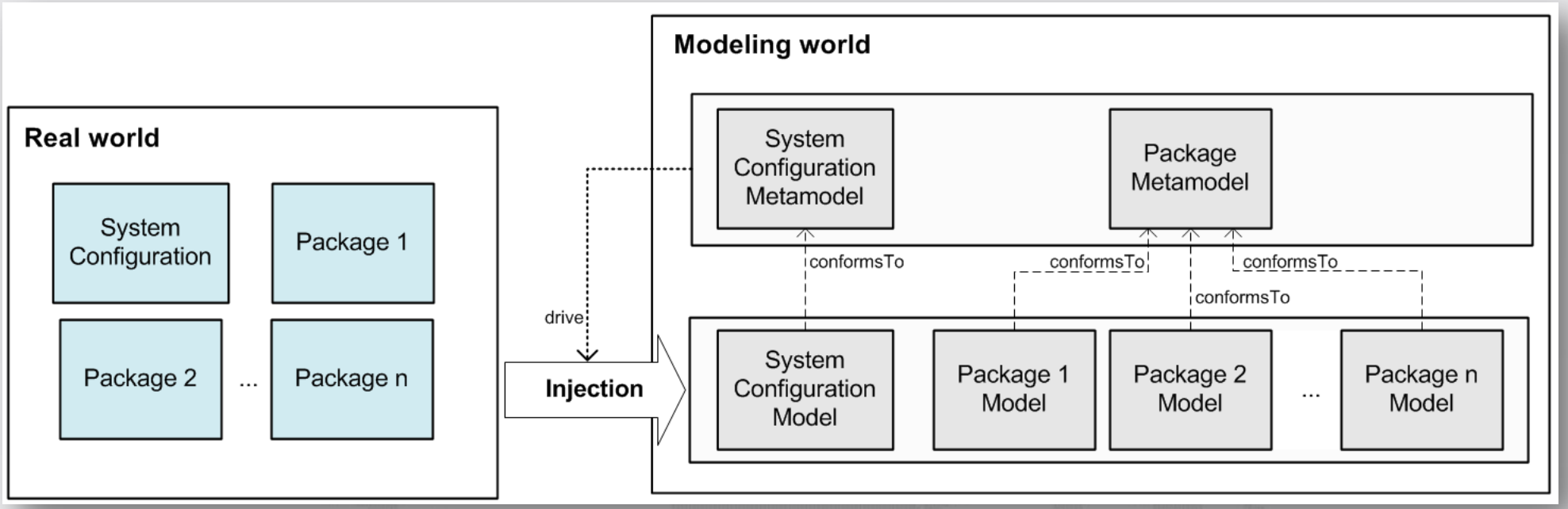
- By means of the model injection, given a real software system a corresponding representation in the modeling world is obtained
- It relies on tools (injectors) that transform software artifacts into corresponding models

Model Injection



- The process is driven by the metamodels
- Difficult to automate completely

Model Injection



» The elements which have to be *injected* are

- The configuration of the real system
- Packages to be installed (maintainer scripts included)

Model Injection: System Configuration

» Existing systems have to be “inspected” in order to generate corresponding models which are defined in terms of the following metaclasses (among others):

- *FileSystem*, to represent the file system by including all the files which build up the configuration (e.g. user files which do not compromise the systems are not taken into account)
- *Init*, used to model the typical `/etc/init.d` location and to maintain the services which have to be started when the system is booted
- *Service*, to model service which are running
- *Alternative*, to model all the existing alternatives. For instance, for the `java` alternative, all the installed versions of the java virtual machine are maintained

Model Injection: System Configuration

» Existing systems have to be “inspected” in order to generate corresponding models which are defined in terms of the following metaclasses (among others):

- *PackageSetting*, for each package a corresponding package setting element is available to refer to its configuration files
- *SharedLibrary*, to model the shared libraries
- *Module*, to represent all the kernel modules
- *User*, to model all the users of the considered system
- *Group*, to model all the groups of the considered system

Model Injection: System Configuration

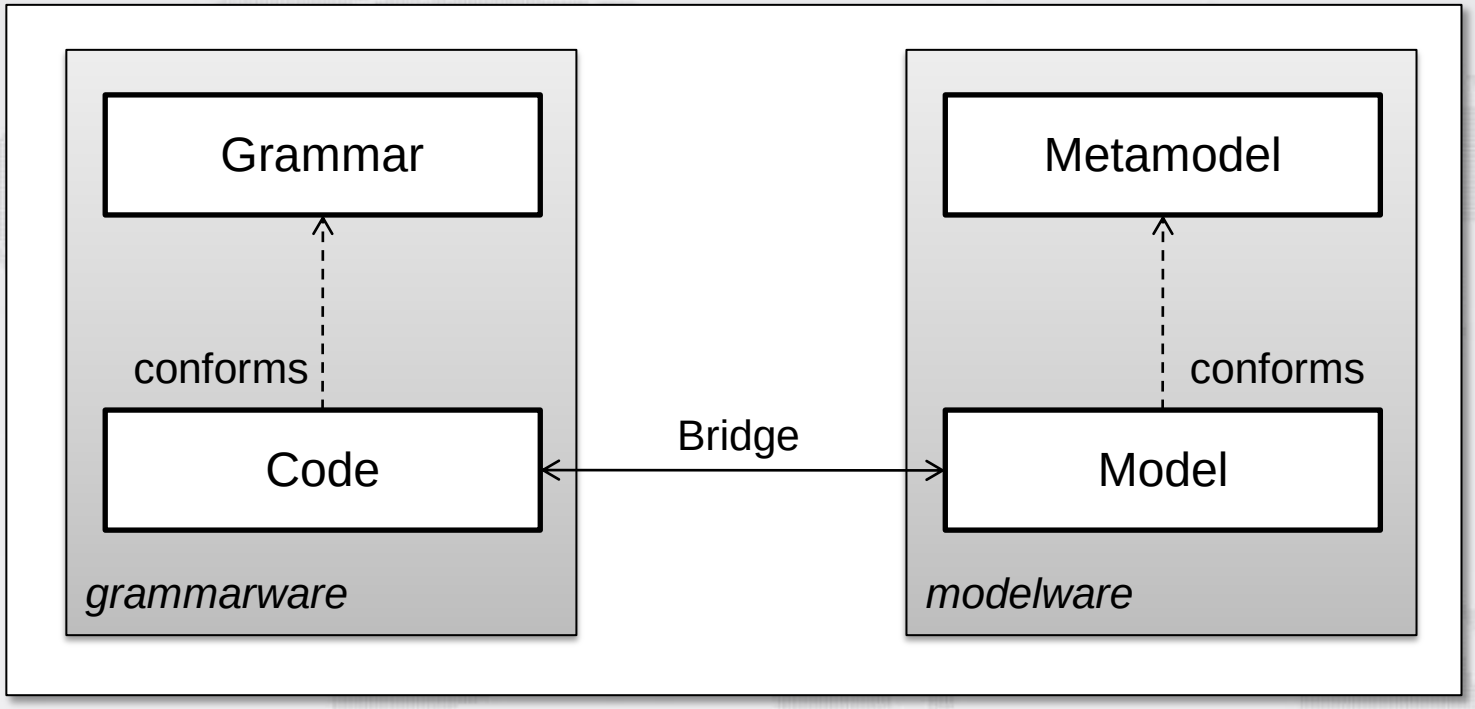
» The generation of system configuration models from existing systems is performed programmatically by using Java and the Eclipse Modeling Framework

» Specific shell commands (like *dpkg-query, ps,* etc.) are invoked by ad-hoc Java programs which parse their results and opportunely create modeling elements

» Open issues:

- Is it possible to extract dependencies among package configurations according to a “general rule” ?
- If not, probably we need kind of incremental dictionary that for each service configuration maintains “possible” dependencies with other package configurations

Model Injection: Maintainer Scripts



Model Injection: Maintainer Scripts

» Several approaches for bridging *grammarware* and *modelware* have been identified

» They can be classified in two groups:

- *Grammar-based* approaches, they generate metamodels from grammars
- *Metamodel-based* approaches, they generate grammars from metamodels

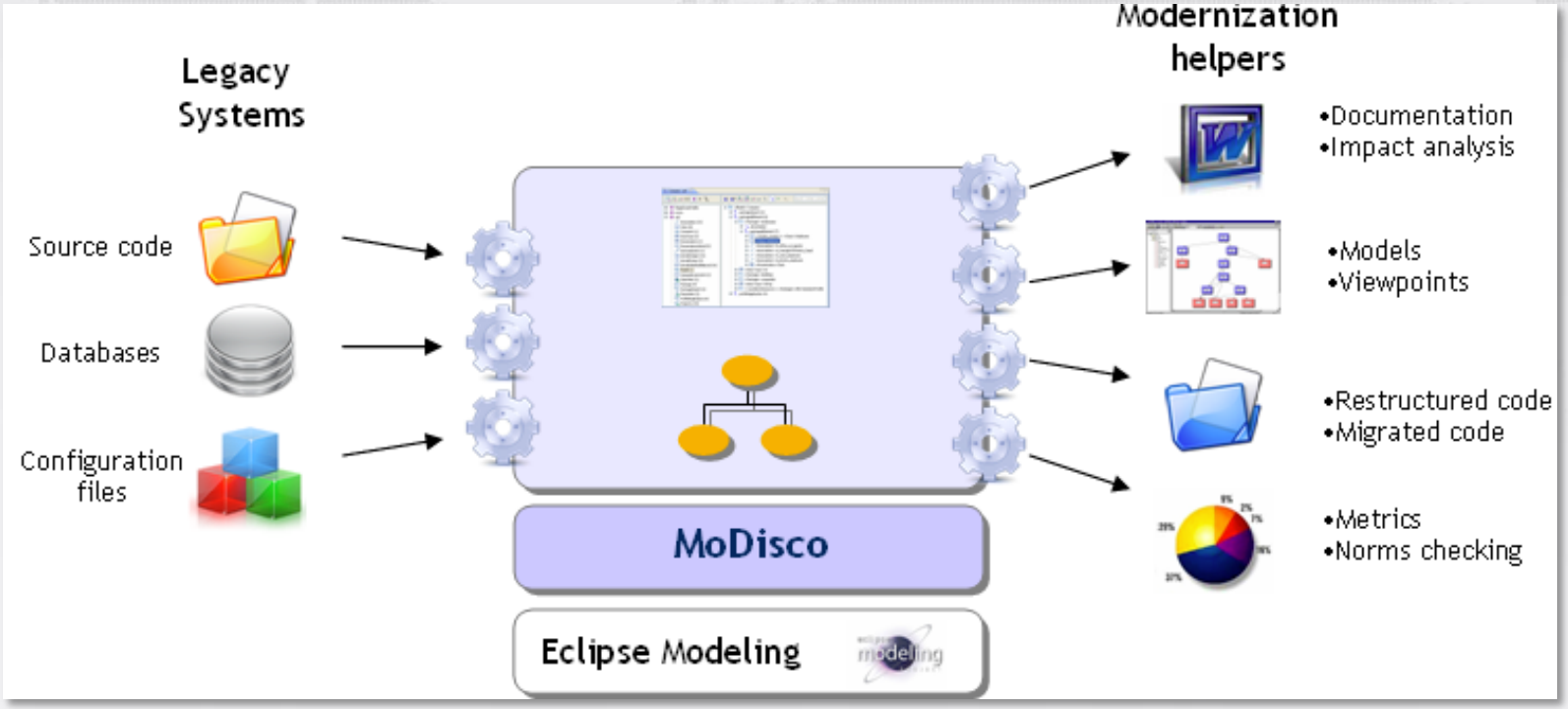
» In model driven software evolution, the process starts from existing source code that conforms to the grammar of a programming language

=> *Metamodel-based* approaches are not well suited, hence *Grammar-based* ones have to be considered

Existing approaches: MoDisco

» It is an attempt in the context of the EU ModelPlex project https://www.modelplex-ist.org/

» It provides an extensible framework to develop model-driven tools to support use-cases of existing software modernization



Existing approaches: MoDisco

» Modernizing an existing software system implies:

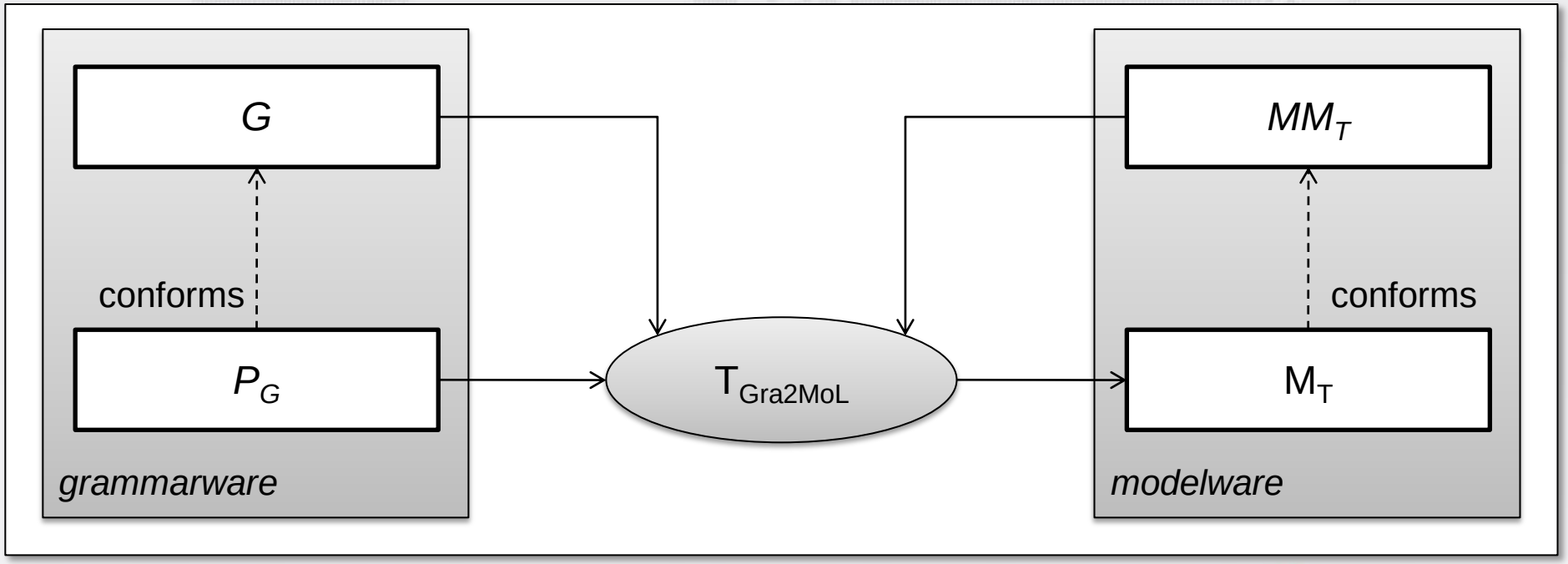
- **Describing** the information extracted out of the artifacts of this system
- **Understanding** the extracted information in order to take the good modernization decisions
- **Transforming** this information to new artifacts facilitating the modernization (metrics, document, transformed code, ...)

» MoDisco aims at supporting these three phases by providing:

- **Metamodels** to describe existing systems
- **Discoverers** to automatically create models of these systems
- **Generic tools** to understand and transform complex models created out of existing systems
- **Use-cases** illustrating how MoDisco can support modernization process

Existing approaches: Gra2MoL

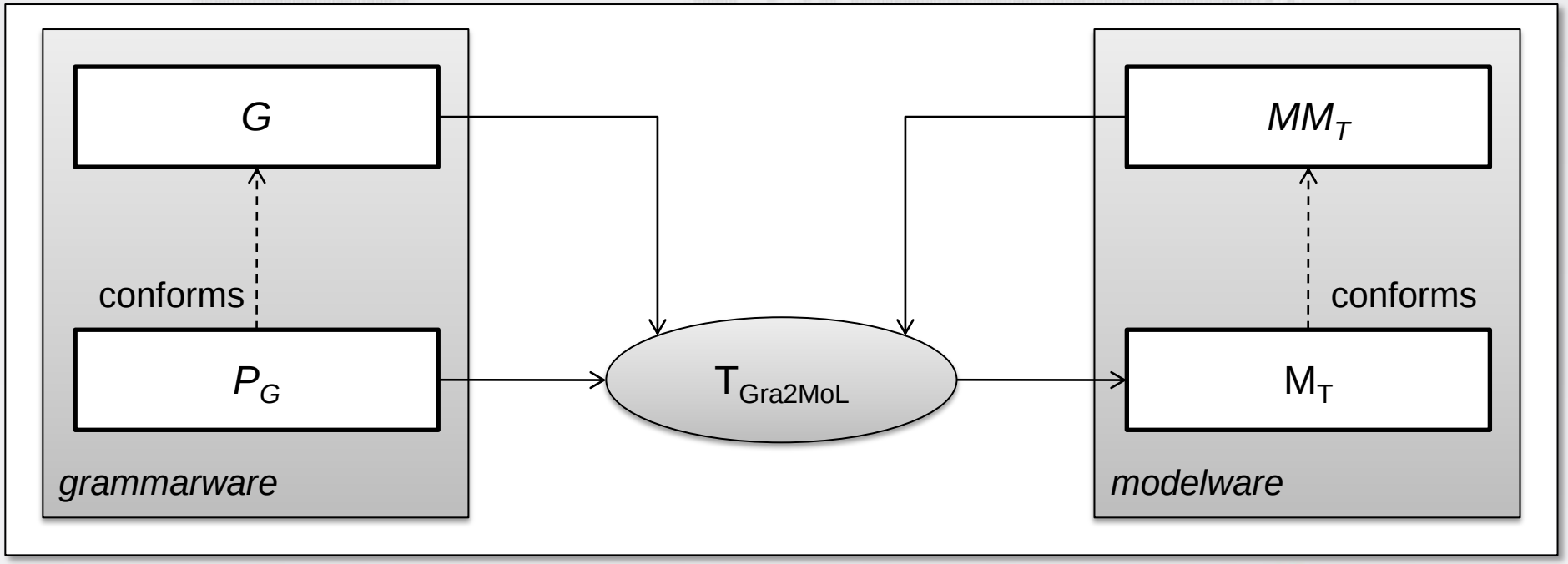
- From the University of Murcia
- It is based on the definition of a grammar-to-model transformation language which is specially tailored to address the grammarware-modelware bridge
- It promotes grammar reuse, and provides domain-specific features such as a query language to traverse syntax trees



Existing approaches: Gra2MoL

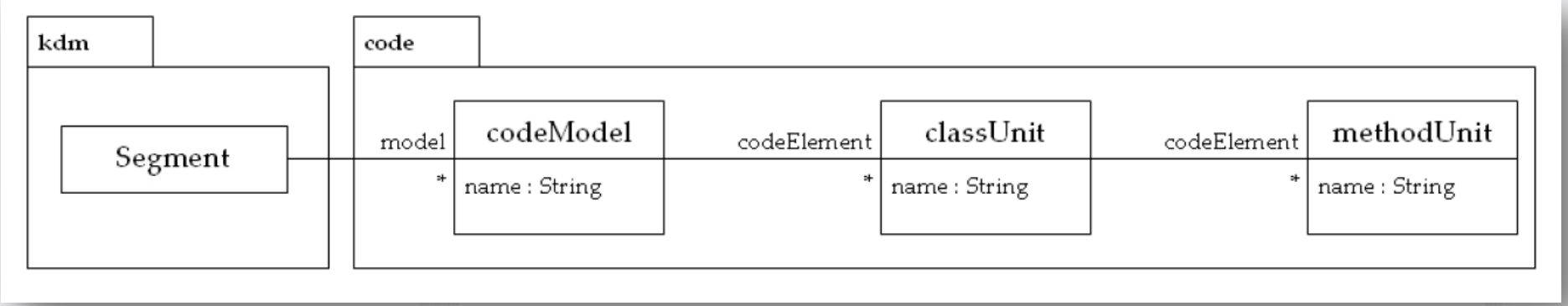
» Given a source program (P_G) conforming to the grammar (G) of a programming language, the objective is to generate a model conforming to a target metamodel MM_T

» The Gra2MoL language is used to explicitly specifying the relationships between source grammar elements and metamodel elements



Gra2MoL example: Java to KDM

- KDM is a metamodel for knowledge discovery in software. It defines a common vocabulary of knowledge related to software engineering artifacts, regardless of the implementation programming language and runtime platform
- KDM is designed as the OMG's foundation for software modernization



Fragment of the KDM metamodel

Gra2MoL example: Java to KDM

```
compilationUnit
 : annotations? packageDeclaration?
   importDeclaration* typeDeclaration*
 ;

typeDeclaration
 : classOrInterfaceDeclaration
 ;

classOrInterfaceDeclaration
 : modifier* (classDeclaration | ...)
 ;

classDeclaration
 : normalClassDeclaration
 | ...
 ;

normalClassDeclaration
 : 'class' classId=Identifier (typeParameters)?
   ('extends' type)?
   ('implements' typeList)?
   classBody
 ;
```

```
classBody
 : '{' classBodyDeclaration* '}'
 ;

classBodyDeclaration
 : modifier* memberDecl
 | ...
 ;

memberDecl
 : methodDeclaration
 | ...
 ;

methodDeclaration
 : type methodName=Identifier
   methodDeclaratorRest
 ;

...
```

Fragment of the Java grammar

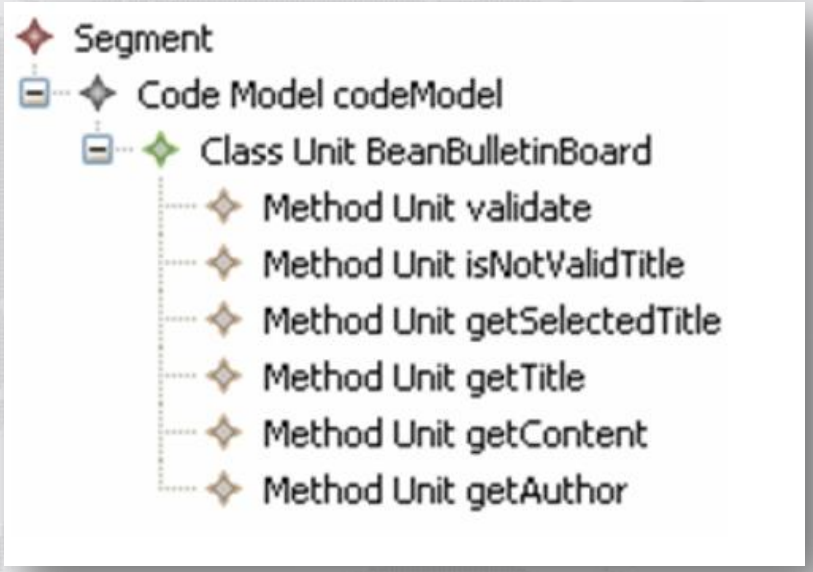
Gra2MoL example: Java to KDM

```
public class BeanBulletinBoard extends ActionForm{

  private String selectedTitle = "";
  private String title = "";
  private String content = "";
  private String author = "";

  public ActionErrors validate(...){...}
  private boolean isNotValidTitle(...){...}
  public void setSelectedTitle(...) {...}
  public String getSelectedTitle() {...}
  public void setTitle(...) {...}
  public String getTitle() {...}
  public void setContent(...) {...}
  public String getContent() {...}
  public void setAuthor(...) {...}
  public String getAuthor() {...}
}
```

Gra2MoL transformation



Gra2MoL example: Java to KDM

```
rule 'createSegment'
  from compilationUnit cu
  to kdm::Segment
  queries
    class : /cu//#normalClassDeclaration;
  mapping
    model = new code::CodeModel;
    model.name = "codeModel";
    model.codeElement = class;
end_rule

rule 'createClass'
  from normalClassDeclaration nc
  to code::ClassUnit
  queries
    ms : /nc//#methodDeclaration[@methodName.exists];
  mapping
    name = nc.classId;
    codeElement = ms;
end_rule

rule 'createMethod'
  from methodDeclaration md
  to code::MethodUnit
  queries
  mapping
    name = md.methodName;
end_rule
```

Fragment of the Gra2MoL Transformation

Gra2MoL example: Maintainer scripts to MANCOOSI

- The (ANTLR) bash grammar is not available
- Two possible solutions can be taken into account
  - Define the ANTLR bash grammar

Gra2MoL example: Maintainer scripts to MANCOOSI

- The (ANTLR) bash grammar is not available
- Two possible solutions can be taken into account
  - Define the ANTLR bash grammar
  - Adopt a “general grammar” like the following

```
grammar bash;

bashDef
    :   commandDef*
    ;

commandDef
    :   commandName param* (PIPE commandDef)* ( ';' | '\n' )
    ;

commandName
    :   ID
    ;
[…]
```

Gra2MoL example: Maintainer scripts to MANCOOSI

- The (ANTLR) bash grammar is not available
- Two possible solutions can be taken into account
  - Define the ANTLR bash grammar
  - Adopt a “general grammar” like the following

This is a very simple grammar definition, even though the main work would fall on the code-to-model transformation definition

```
grammar bash;

bashDef
    :   commandDef*
    ;

commandDef
    :   commandName param* (PIPE commandDef)* ( ';' | '\n' )
    ;

commandName
    :   ID
    ;
[…]
```

To summarize

- The deliverable D2.2 will contain
  - An introduction to the proposed model driven approach to support the upgrade of FOSS systems
  - Introduction to the model injection problem
  - How to deal with model injection on a particular FOSS system
  - Existing tools will be outlined
  - The injection of the system configuration is presented
  - The injection of the maintainer scripts based on the Gra2MoL approach will be also presented

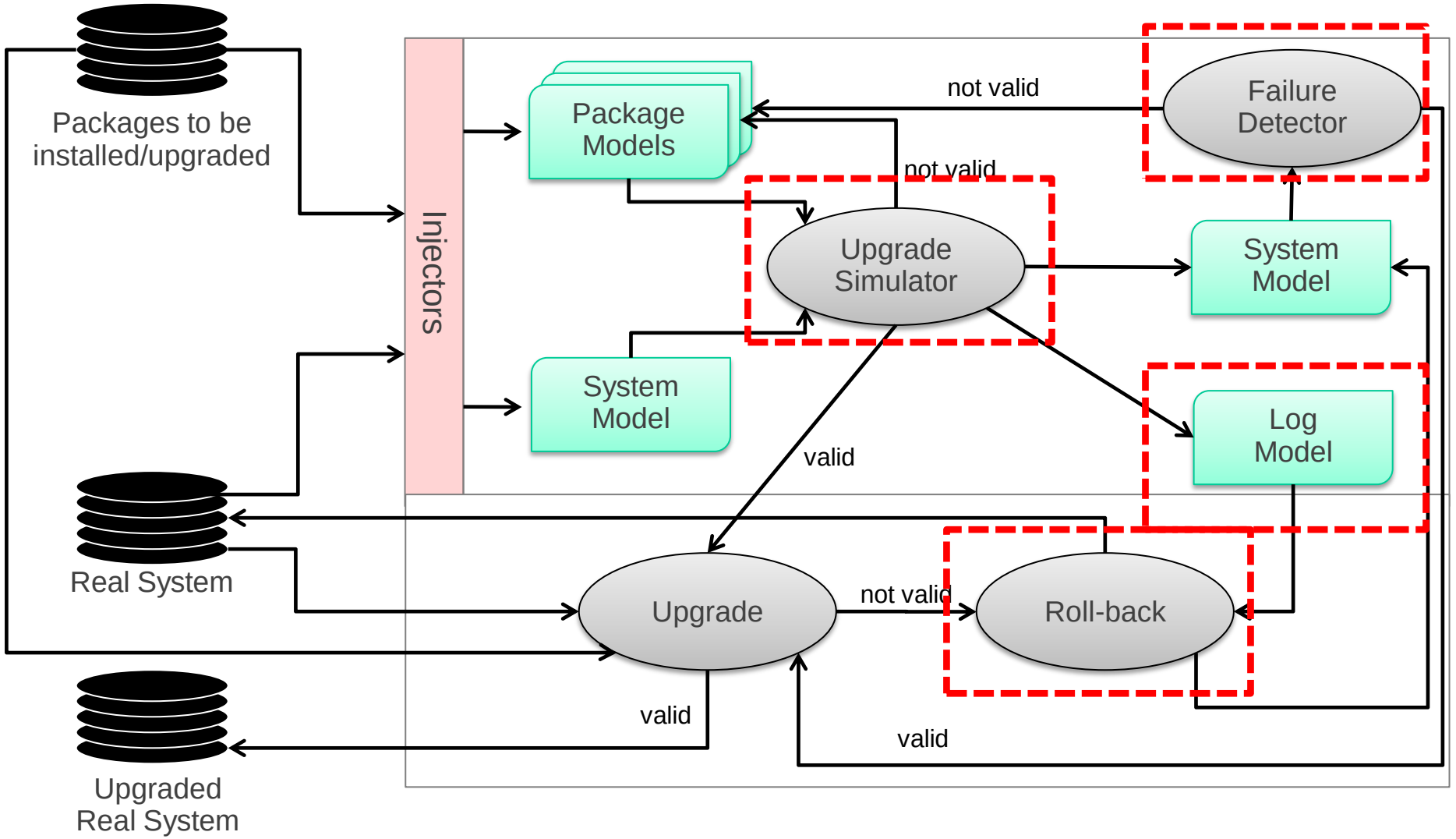
Outline

- Introduction
- Instantiation of the metamodel on a wide-used GNU/Linux distribution
  - Model Injection
  - Supporting tools
- Next steps
  - Model-based framework for managing the complexity and the state of the GNU/Linux instantiation
- Conclusions

Next steps

- Model-based framework for managing the complexity and the state of the GNU/Linux instantiation
- This is due at T0+36 and it will consist of
  - Simulator
  - Failure detector
  - …
- Many collaboration points have to be discussed by taking into account the overall model driven approach

Next steps


Packages to be installed/upgraded
Injectors
Package Models
not valid
Failure Detector
not valid
Upgrade Simulator
System Model
System Model
valid
Log Model
Real System
Upgrade
not valid
Roll-back
valid
valid
Upgraded Real System

References

- MoDisco Home page, http://www.eclipse.org/gmt/modisco/
- J.L.C. Izquierdo, J.S. Cuadrado, J.G. Molina, **Gra2MoL: A domain specific transformation language for bridging grammarware to modelware in software modernization**, MODSE 2008. Workshop on Model Drvine Software Evolution
- Gra2MoL Home page: http://modelum.es/gra2mol

Towards D2.2 and WP2 next steps

Davide Di Ruscio – University of L'Aquila