BEYONDwork: A Model-Driven Development Environment for Biologically-inspired Autonomic Network Applications

Hiroshi Wada[1], Chonho Lee[1], Junichi Suzuki[1] and Tetsuo Otani[2]

[1] University of Massachusetts, Boston MA 02125, USA,
{shu, chonho, jxs}@cs.umb.edu
[2] Central Research Institute of Electric Power Industry, Komae-Shi, Tokyo 201-8511, Japan
ohtani@criepi.denken.or.jp

Abstract. This paper proposes a model-driven development (MDD) environment, called BEYONDwork, which addresses a key software engineering issue in autonomic computing: the complexity of designing and maintaining operational policies. BEYONDwork provides (1) a set of visual domain-specific languages (DSLs) to graphically model the operational policies in autonomic network applications, and (2) a supporting tool that validates the operational policies defined in the proposed DSLs and transform them to program code. BEYONDwork also addresses an issue in DSLs: low flexibility for customization. Through the notion of Multi-Staged Model Reflection (MSMR), it allows users to graphically customize DSL's grammar (or metamodel) in an intuitive manner. This paper describes the design and implementation details of BEYONDwork.

1 Introduction

A key software engineering issue in autonomic computing is the complexity of designing and maintaining operational policies, which describe a set of administrative decisions for application operations. It is often tedious and time-consuming for application administrators to design proper operational policies because policy design involves a large volume of information. Administrators need to consider a variety of operational conditions (e.g., application's internal states and external environment conditions such as resource consumption and workload) and administrative actions (e.g., migration of an application from one host to another). Due to this information overloading, administrators can be overwhelmed to pair operational conditions and administrative actions.

It is also expensive and error-prone to maintain operational policies when an application's functional and/or non-functional aspects often change. Functional changes include introducing new application functionalities and changing existing ones. Non-functional changes include updating service level agreements and adding new hardware resources. Upon these changes, administrators need to consider additional operational conditions and administrative actions and re-define the pairs of conditions and actions.

Domain-specific modeling is a promising solution for administrators to design and maintain the operational policies of autonomic applications. In general, domain-specific modeling provides visual domain-specific languages (DSLs) that directly capture, represent and implement the concepts in particular problem domains, rather than general-purpose languages that are aimed at any software problems [1, 2]. A visual DSL represents each domain-specific concept as its modeling primitive. This simplifies the process to build visual models and allows users (even non-programmers) to understand, design and maintain models. Moreover, DSLs intentionally limit their expressiveness to specify a particular set of domain-specific concepts; therefore, they can reduce the chances for users to make errors by building models in invalid or unexpected ways.

This paper proposes a model-driven development (MDD) environment, called BEYONDwork, which consist of (1) a set of visual DSLs to model the operational policies in autonomic network applications, and (2) a supporting tool for the DSLs. The proposed DSLs specialize in the operational policies (i.e., operational conditions and administrative actions) in autonomic network applications, and defines those domain-specific concepts as their language grammars (or metamodels). The DSLs allows application administrators (i.e., non application programmers) to graphically design and maintain the operational policies in an intuitive manner. The proposed tool validates the operational policies defined in the proposed DSLs, and transform them to program code. This code generation enables rapid configuration of autonomic applications.

The proposed MDD environment also addresses an issue in domain-specific modeling: low flexibility for customization. Currently, DSLs assume that their grammars (or metamodels) are stable. As far as a DSL's metamodel does not change, users (e.g., application administrators) can enjoy the DSL's ease of use. However, it is very hard, if not impossible, for them to change the DSL's metamodel for supporting new domain-specific concepts or changing the semantics of existing domain-specific concepts. This customization requires the knowledge on the DSL's design details; how the syntax and semantics of domain-specific concepts are defined as a metamodel with a meta-metamodeling language (e.g., Eclipse Modeling Framework; www.eclipse.org/emf). DSL users usually do not have such knowledge. BEYONDwork addresses this issue through the notion of *Multi-Staged Model Reflection (MSMR)*. In MSMR, a metamodel of a DSL is customized with a model of another DSL. Without requiring the knowledge on meta-metamodels, MSMR allows DSL users to graphically customize DSLs.

2 BEYOND: An Architecture for Autonomic Networking

BEYONDwork is currently intended to support an architecture for autonomic network applications, called BEYOND[3] This section briefly overviews BEYOND to better explain BEYONDwork in Section 3. See [3] for the full discussion on BEYOND.

BEYOND is designed to address two issues in autonomic network applications: *autonomy*–the ability to operate without intervention to/from human administrators, and *adaptability*–the ability to adapt to dynamic environment conditions in the network (e.g., network traffic and resource availability). Inspired by an observation that various biological systems have developed the mechanisms to overcome these issues, BEYOND applies key biological principles and mechanisms to design network applications.

[3] Biologically-Enhanced sYstem architecture beyond Ordinary Network Designs.

2.1 Agents

In BEYOND, each network application is designed as a decentralized group of software agents. This is analogous to a bee colony (application) consisting of multiple bees (agents). Each agent provides a certain functionality of an application, and implements biologically-inspired behaviors. Each agent also carries its own *behavior policy*, which determines which behavior to be invoked in a given set of environment conditions. Behavior policies in BEYOND are equivalent to operational policies in autonomic computing. Example agent behaviors are listed below.

- **Energy exchange and storage:** Biological entities strive to seek and consume food for living. In BEYOND, agents store and expend *energy* for living. Each agent gains energy in exchange for performing its service to other agents or human users, and expends energy to use the resources available at the local host (e.g., memory space).
- **Replication:** Agents may make their copies in response to high energy level, which indicates high demand for the agents. A replicated agent is placed on the host that its parent agent resides on, and it inherits the parent's behavior policy. Mutation may occur on the inherited behavior policy.
- **Reproduction:** Agents may reproduce child agents with other agents (mating partners). A child agent is placed on the host that its parents reside on, and it inherits behavior policies from both parents through crossover. Mutation may occur on the behavior policy of a child agent.
- **Migration:** Agents may move from one network host to another.
- **Death**: Agents die due to energy starvation. If an agent cannot balance its energy expenditure with its energy gain, the agent cannot pay for the resources it needs; thus, it dies from lack of energy.

2.2 iNet: Agent Adaptation Mechanism in BEYOND

iNet is a key component in BEYOND, which allows each agent to adaptively perform its behaviors against dynamic environment conditions in the network. iNet is designed after the mechanisms behind how the immune system detects antigens (e.g., viruses), how it specifically produces antibodies to eliminate them, and how it evolves antibodies to react to a massive number of antigens. iNet models a set of environment conditions as an antigen and an agent behavior as an antibody. Each agent contains its own immune system, and a configuration of the agent's antibodies defines its behavior policy. iNet allows each agent to autonomously sense its surrounding environment conditions (i.e., antigen) for evaluating whether it adapts well to the sensed conditions, and if it does not, adaptively invoke a behavior (i.e., antibody) suitable for the conditions. For example, agents may invoke the replication behavior at the network hosts that accept a large number of user requests for their services. This leads to the adaptation of agent availability; agents can improve their throughput. Also, agents may invoke the migration behavior to move toward the network hosts that receive a large number of user requests for their services. This results in the adaptation of agent locations; agents can improve their response time to user requests.

2.2.1 Natural Immune System The natural immune system adaptively regulates the body against dynamic environmental changes such as antigen invasions. Through a number of interactions among various white blood cells (e.g., macrophages and lymphocytes such as T-cells and B-cells) and molecules (e.g., antibodies), the immune system evokes two responses to antigens: *T-cell activation* and *B-cell activation* responses.

In the T-cell activation response, the immune system performs self/non-self discrimination. This response is initiated by macrophages. Macrophages move around the body to ingest antigens and present them to T-cells. T-cells are produced in thymus though the negative selection. In this selection process, thymus removes T-cells that strongly react to the body's own (self) cells. The remaining T-cells are used as detectors to identify foreign (non-self) cells. When a T-cell detects a non-self cell presented by a macrophage, the T-cell secretes chemical signals to induce the B-cell activation response.

In the B-cell activation response, B-cells are activated by T-cells. Some of the activated B-cells strongly react to an antigen, and they produce antibodies that specifically kill the antigen. Antibodies form a network and communicate with each other [4]. This immune network is formed with stimulation and suppression relationships among antibodies. With these relationships, antibodies dynamically change their populations and network structure. For example, the population of specific antibodies rapidly increases following the detection of an antigen and, after eliminating the antigen, decreases again.

The immune system maintains approximately 10^9 antibodies. B-cells can increase this repertoire further by mutating and recombining immune gene segments so that antibodies can bind a massive number of antigens [5].

2.2.2 iNet Artificial Immune System The iNet artificial immune system consists of the environment evaluation (EE) facility and behavior selection (BS) facility, which implement the T-cell and B-cell activation responses, respectively (Figure 1). The EE facility allows an agent to continuously sense a set of current environment conditions as an antigen and classify the antigen to self or non-self. A self antigen indicates that the agent adapts to the current environment conditions well, and a non-self antigen indicates it does not. When the EE facility detects a non-self antigen, it activates the BS facility. The BS facility allows an agent to choose a behavior as an antibody that specifically matches with the detected non-self antigen.

The EE facility performs two steps: initialization and self/non-self classification. The initialization step produces detectors (i.e., T-cells) that identify self and non-self antigens. Each antigen is represented as a feature vector (X), which consists of a set of environment conditions, or features, (F_i) and a class value (C): $X = (F_1, F_2, ..., F_n, C)$.

C indicates whether a given antigen (i.e., a set of environment conditions) is self (0) or non-self (1). If an agent senses resource utilization and workload (the number of user requests) on the local host, an antigen is represented like $X_{current} = ((Low : ResourceUtilization, Low : Workload), 0)$.

The initialization of the EE facility is designed after the negative selection in the immune system (Figure 2). As the immune system randomly generates T-cells first, the EE facility generates detectors (feature vectors) randomly. Then, the EE facility separates the detectors into self detectors, which closely match with self antigens, and non-self detectors, which do not. This separation is performed via vector similarity

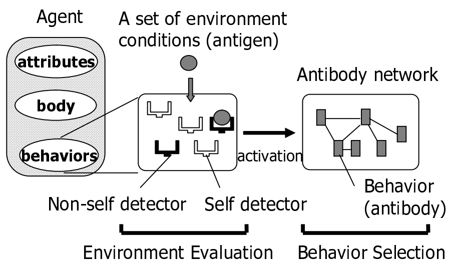


Fig. 1: iNet Architectural Design

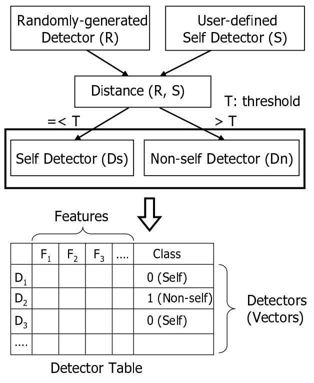


Fig. 2: Initialization of the EE Facility

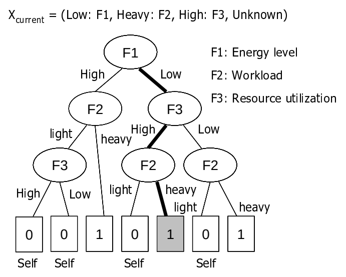


Fig. 3: An Example Decision Tree

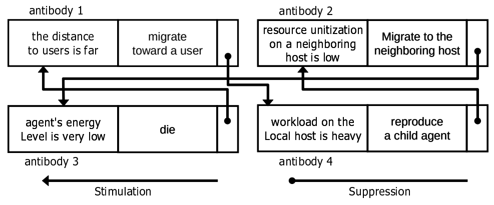

antibody 1
the distance to users is far
migrate toward a user
antibody 2
resource unitization on a neighboring host is low
Migrate to the neighboring host
agent's energy Level is very low
die
antibody 3
workload on the Local host is heavy
reproduce a child agent
antibody 4
Stimulation
Suppression

Fig. 4: An Example Antibody Network

measurement between randomly generated feature vectors (R) and self antigens (S) that human administrators supply. After the vector matching, both self and non-self detectors are stored in the detector table (Figure 2)[4].

In self/non-self classification, the EE facility classifies a given antigen to self or non-self. This is performed with a decision tree built from the detectors in the detector table. Figure 3 shows an example decision tree. Each node in the tree specifies which feature (environment condition) is considered. Based on the feature values in a given antigen, the EE facility travels through tree branches. If the EE facility classifies the antigen to non-self, it activates the BS facility.

The BS facility selects an antibody (i.e., agent's behavior) suitable for a detected non-self antigen (i.e., a set of environment conditions). Each antibody consists of three parts: a *precondition* under which it is selected, *behavior ID* and *relationships* to other antibodies. Antibodies are linked with each other using stimulation and suppression relationships. Each antibody has its own concentration value, which represents its population. The BS facility identifies candidate antibodies suitable for a given non-self antigen, prioritizes them based on their concentration values, and selects the most suit-

[4] The immune system removes non-self detectors through negative selection. However, in iNet, both self and non-self detectors are used to perform self/non-self classification.

able one from the candidates. When prioritizing antibodies, stimulation relationships among them contribute to increase their concentration values, and suppression relationships contribute to decrease it. Each relationship has an affinity value, which indicates the degree of stimulation or suppression.

Figure 4 shows an example network of antibodies. It contains four antibodies, which represent the migration, replication and death behaviors. Antibody 1 represents the migration behavior invoked when the distance to users is far from an agent. Antibody 1 suppresses Antibody 3 and stimulates Antibody 4. Now, suppose that a (non-self) antigen indicates (1) the distance to users is far, (2) workload is heavy on the local host and (3) resource utilization is low on a neighboring platform. This antigen stimulates Antibodies 1, 2 and 4 simultaneously. Their populations increase, and Antibody 2's concentration value becomes highest because Antibody 2 suppresses Antibody 4, which in turn suppresses Antibody 1. As a result, the BS facility would select Antibody 2.

As Section 2.2.1 describes, the immune system diversifies antibodies by mutating immune genes so that antibodies can react to unanticipated antigens. Similarly, iNet diversifies antibodies via gene operations such as mutation and crossover so that agents can adapt to unanticipated environment conditions. In iNet, each agent encodes and possesses its own antibody network configuration (behavior policy) as a set of genes. When a new agent is born through a replication or reproduction process, it interprets the genes given by its parent(s) and form an antibody network.

3 BEYONDwork: Agent Development Environment in BEYOND

BEYONDwork is an application development environment for iNet. It provides two visual modeling DSLs and a textual programming DSL for configuring environment conditions, detectors and behavior policies. Figure 5 shows the iNet configuration process with BEYONDwork. BEYONDwork consists of five facilities: environment configuration facility (Section 3.1), agent behavior configuration facility (Section 3.2), EE configuration facility (Section 3.3), BS configuration facility (Section 3.4) and code generator. Each facility except the code generator is a visual or textual development tool specialized to a certain purpose, i.e., domain-specific model. By providing a series of modeling/textual DSLs and supporting tools specialized to certain purposes, BEYONDwork intentionally limits the level of expressiveness of their users (domain experts). It prevents users from defining invalid models (configurations).

The environment configuration facility allows environment condition designers to configure environment conditions with a visual DSL. The agent behavior configuration facility allows agent behavior designers to configure agent behaviors (see Section 2.1) with a visual DSL. The EE configuration facility allows agent designers to configure a set of detectors to identify self and non-self antigens (Section 2.2.2) based on environment conditions configured in the environment configuration facility. The BS configuration facility allows agent designers to configure their agents' behavior policies (antibody configuration) with visual or textual DSLs. Both DSLs have the same level of expressiveness, and the artifacts of the DSLs (models and programs) are transparently translatable with each other. Agent designers can configure the behavior policy of each agent through the use of either DSL.

The EE configuration facility and the BS configuration facility are configured to support environment conditions defined in the environment configuration facility and agent behaviors defined in the agent behavior configuration facility. Especially, the BS configuration facility is customized through the notion of Multi-Staged Model Reflection (MSMR).

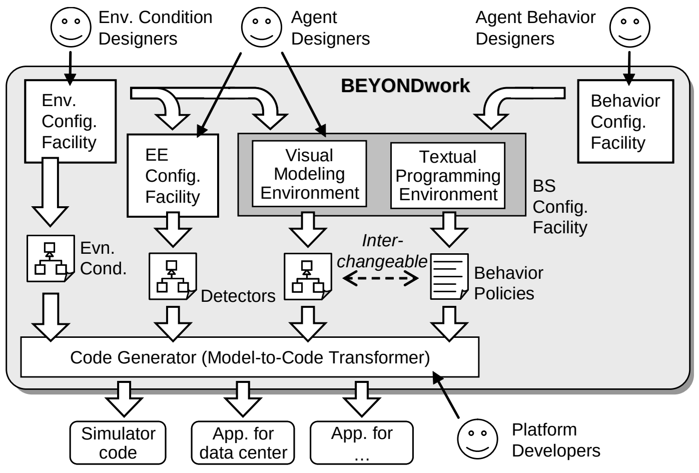


Fig. 5: iNet Configuration with BEYONDwork

Once environment conditions, agent behaviors, detectors and a behavior policy are complete in the form of visual models or textual programs, the code generator transforms them to compilable source code by following a transformation rule between the languages and source code. The transformation rules are implemented by platform developers, who know the details of platform technologies. (e.g., operating systems, middleware, simulators and programming languages) Through changing one transformation rule to another, the code generator can generate source code that are compatible with different deployment environments such as simulators and real networks. Environment condition designers and agent designers do not have to write different models/programs for the same agent running on different platform technologies. This flexible code generation feature improves the productivity of agent designers. Currently, BEYONDwork supports Java code generation for a simulator in BEYOND.

3.1 BEYONDwork Environment Configuration Facility

Figure 6 shows an example environment configuration model which defined in the environment configuration facility. As Figure 6 illustrates, the visual language visualizes an environment condition as a rectangle. Each rectangle can contain arbitrary number of rounded rectangles representing categories of a corresponding environment condition. For example, in Figure 6, the `LocalWorkload` environment condition has two categories, `HEAVY` and `LIGHT`. Also, each category specifies a condition to classify a corresponding environment condition. In iNet, each environment condition is supposed to have one representative value, e.g., workload or the number of agents, and the representative value is used to identify the category of a corresponding environment condition. For example, in Figure 6, the `LocalWorkload` environment condition is classified as `HIGH` when its representative value exceeds 200, otherwise classified as `LIGHT`.

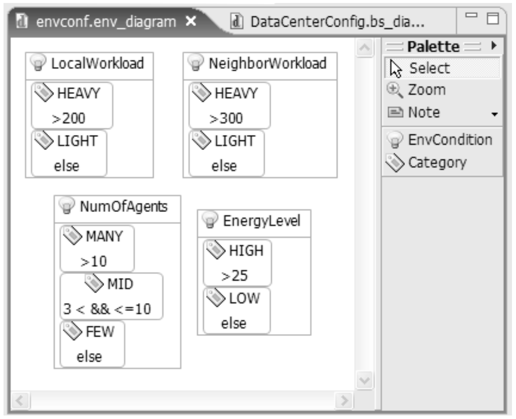


Fig. 6: BEYONDwork Environment Configuration Facility

The details of representative values, e.g., how and when to obtain the value, are hidden from environment condition designers and agent designers. Platform developers implement such details on skeleton code generated by the code generator. For example, Listing 1.1 is a fragment of Java source code generated from the `LocalWorkload` environment condition in Figure 6. The class `edu.umb.inet.simulator.EnvironmentCondition` is a parent class for environment conditions working on a simulator, and it provides a facility to access to states of a simulator, e.g., a request rate and CPU load. Platform developers implements the `getRepValue` method by leveraging APIs provided by `edu.umb.inet.simulator.EnvironmentCondition`, e.g., returns a request rate, CPU load, or the summation of them, a representative value of an environment condition is retrieved and evaluated against conditions specified by each category.

Listing 1.1: An Example Generated Code

```java
public class LocalWorkload
  extends edu.umb.inet.simulator.EnvironmentCondition
  implements EnvironmentCondition {

  enum Category{ HIGH, LOW };
  public Category evaluate(){
    double repValue = getRepValue();
    if( repValue > 200 ){ return Category.HIGH; }
    else{ return Category.LOW; }
  }
  private double getRepValue(){
    // TODO: platform developers add code here
  }
}
```

Figure 7 shows the meta-model of environment configuration models. Any environment configuration model, e.g., Figure 6, is defined as an instance of this metamodel, and any subsequent tools, e.g., a metamodel generator for BS configuration facility and code generator, access environment configuration model through this metamodel.

The metamodel has three metaclasses, `EnvironmentConditions`, `EnvironmentCondition` and `Category`. `EnvironmentConditions` represents a set of environment conditions. This metaclass has no corresponding graphical notation since its instance is an environmental configuration model itself. An instance of `EnvironmentCondition` metaclass represents an environment condition, e.g., `LocalWorkload` in Figure 6, and its `name` attribute is a name of an instance. An instance of `EnvironmentCondition` metaclass can contains arbitrary number of instances of `Condition` metaclasses, which represents a category of an environment condition. The `name` attribute is a name of an instance, and the `expression` attribute maintains a string representing a condition, e.g., `"> 200"` in the `Heavy` category of the `LocalWorkload` in Figure 6.

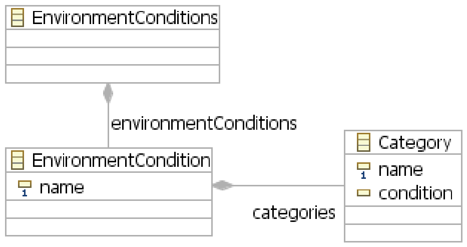


Fig. 7: Metamodel of Environment Configuration Model

The metamodel for environmental configuration models is defined in Eclipse Modeling Framework (EMF; http://www.eclipse.org/emf/), and the environment configuration facility is implemented on Eclipse Graphical Modeling Framework (GMF; http://www.eclipse.org/gmf/). GMF takes an metamodel defined in EMF as an input, and generate a skeleton for a GUI application[5] which is specialized to the input metamodel.

A transformation from visual models and Java source code running on a simulator is implemented with a model-code transformation engine in openArchitectureware (oAW; http://www.openarchitectureware.org/), an open-source MDD tool. Listing 1.2 is a fragment of the transformation rule.

Listing 1.2: A Fragment of Transformation Rule

```
<<DEFINE ExpandEnvCondition FOR EnvironmentCondition>>
  <<FILE name + ".java">>
  public class <<name>>
    extends edu.umb.inet.simulator.EnvironmentCondition
    implements EnvironmentCondition {

    enum Category{
      <<EXPAND RetrieveCategoryName FOREACH (List[Category])ownedAttribute>> };
    public Category evaluate(){
      double repValue = getRepValue();
      <<EXPAND ExpandCondition FOREACH (List[Category])ownedAttribute>>
    }
    ...
  }
  <<ENDFILE>>
```

[5] Application developers need to define notations for each metamodel elements by hand

```
16 <<ENDDEFINE>>
17
18 <<DEFINE RetrieveCategoryName FOR Category>>
19   <<name>>,
20 <<ENDDEFINE>>
21
22 <<DEFINE ExpandCondition FOR Category>>
23   // transform a category to a if statement
24   // according to its condition
25 <<ENDDEFINE>>
```

A rule for a certain metaclass is defined by the keyword `DEFINE` with the rule's name (line 1). Each instance of `EnvironmentCondition` is transformed into a Java class of which name is the same as the instance's name. (`<<name>>` will be replaced with a name of an instance.) A Java enumeration type is defined according to categories defined in a environment condition (line 7). The enumeration type is completed by calling the `RetrieveCategoryName` rule (defined from line 18 to 20) and set the names of categories as its elements. The `evaluate()` method is also completed by calling the `ExpandCondition` rule on each categories.

By changing a rule to apply, e.g., rules for different platform technologies, an appropriate code is generated without changing an input environment configuration model. It makes easy to reuse an input model and implement applications on different platform technologies.

3.2 BEYONDwork Agent Behavior Configuration Facility

Figure 8 shows an example agent behavior configuration model which defined in the environment configuration facility. As Figure 8 illustrates, the visual language visualizes an agent behavior as a rectangle. Each rectangle can contain arbitrary number of rectangles representing parameters of a corresponding agent behavior. For example, in Figure 8, the `Reproduction` behavior has three parameters, `mutationRate`, `partnerSelectionPolicy` and `crossoverPolicy`. Each parameter consists of its name and data type. The agent behavior configuration facility allows defining enumeration types, e.g., `PartnerSelectionPolicy` in Figure 8

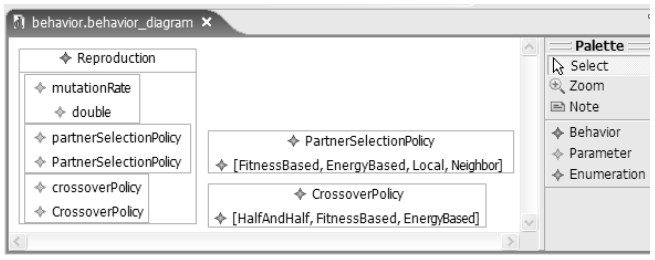


Fig. 8: BEYONDwork Agent Behavior Configuration Facility

Figure 9 shows the meta-model of agent behavior configuration models. The meta-model consists of four metaclasses, `Behaviors`, `Behavior`, `Parameter` and `Enumeration`. `Behaviors` represents a set of behaviors, i.e., agent behavior configuration model itself. An instance of `Behavior` metaclass represents an agent behavior, e.g., `Reproduction` in

Figure 8, and it can contain arbitrary number of instances of `Parameter` metaclasses, which represents parameters.

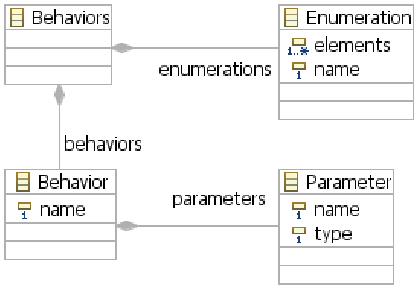
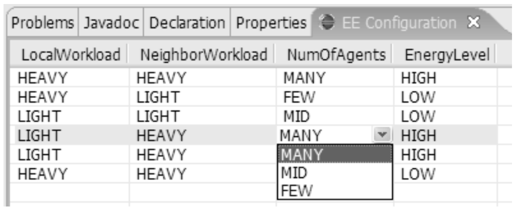


Fig. 9: Metamodel of Behavior Configuration Model

As well as the environmental configuration facility, the metamodel for agent behavior configuration models is defined in EMF and the agent behavior configuration facility is implemented on GMF.

3.3 BEYONDwork EE Configuration Facility

Figure 10 shows the EE configuration facility appears as one of windows in BEYONDwork, located below the BS configuration facility (Figure 12). Each column in the table represents an environment condition configured in the environment configuration facility (Section 3.1) and each row represents a detector. The EE configuration facility allows agent designers to add and remove detectors, and configure detectors by selecting the categories of each environment condition. For example, in Figure 6, the `NumOfAgents` environment condition is configured to have three categories, `MANY`, `MID` and `FEW`, and cells in the corresponding column in Figure 10 allows agent designers to select its value from `MANY`, `MID` and `FEW`. From a set of detectors, BEYONDwork automatically create a decision tree and deploys it in agents (see Section 2.2.2).



LocalWorkload	NeighborWorkload	NumOfAgents	EnergyLevel
HEAVY	HEAVY	MANY	HIGH
HEAVY	LIGHT	FEW	LOW
LIGHT	LIGHT	MID	LOW
LIGHT	HEAVY	MANY	HIGH
LIGHT	HEAVY		HIGH
HEAVY	HEAVY		LOW

Fig. 10: BEYONDwork EE Configuration Facility

The EE configuration facility configures itself, i.e., names of columns and values each sell can select, by retrieving information on environment conditions and their categories from an environment configuration model defined in the environment configuration facility (Section 3.1) through the use of the metamodel of environment configuration model (Figure 7).

3.4 BEYONDwork BS Configuration Facility

Figure 12 and 13 show the visual modeling and textual programming environments in the BS configuration facility, respectively. They are configured to support environment conditions defined in the environment configuration facility (Section 3.1) and agent behaviors defined in the agent behavior configuration facility (Section 3.2) through the notion of MSMR, i.e., an environment configuration model and an agent behavior model are used as a part of the metamodel in the BS configuration facility (Figure 11). This mechanism allows users, even domain experts, to customize a DSL without knowing the details of the DSL's metamodel.

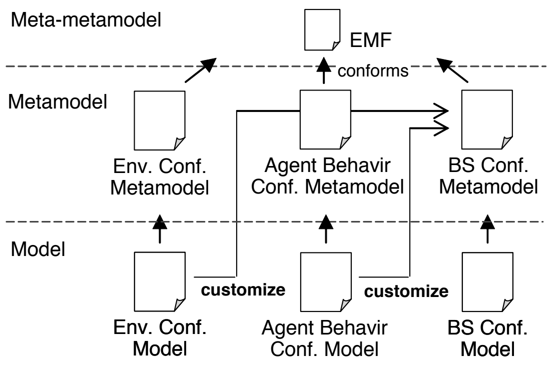
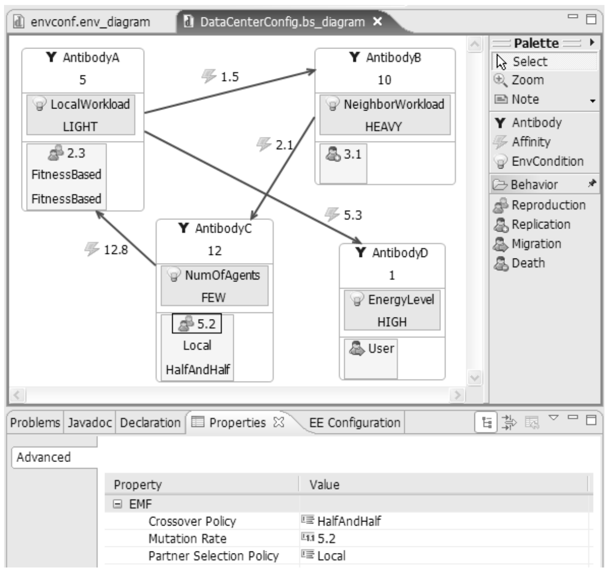


Fig. 11: Multi-Staged Metamodel Reflection in BEYONDwork

As Figure 12 illustrates, the visual language visualizes an antibody as a rounded rectangle. Each rectangle consists of three compartments: (1) the name and the initial concentration of an antibody, (2) an environment condition to which an antibody reacts, and (3) an agent behavior and its properties. For example, in Figure 12, `AntibodyA`'s initial concentration value is 5, and it represents the reproduction behavior. The behavior is invoked when `LocalWorkload` is high. A stimulation/suppression relationship between antibodies is visualized as an solid arrow between rounded rectangles. Each arrow has value, which represents affinity value of a relationship. As Figure 12 demonstrates, the visual language supports all the concepts in antibody configurations as built-in model elements, and agent designers can configure antibodies (agent behavior policies) in an intuitive and rapid manner.

In the textual language (Figure 13), each antibody is defined with the built-in keyword `antibody`. The program in Figure 13 and the model in Figure 12 define the semantically same antibody configuration. As Figure 13 shows, the textual programming environment in BEYONDwork shows built-in keywords in boldface, automatically performs a syntax check, and reports syntax errors while antibody designers configure antibodies. In Figure 13, a syntax error is reported as a cross mark. (The textual language does not support keyword `energylevel` but `EnergyLevel` because of the environment conditions defined in Figure 6.)

Listing 1.3 is a fragment of Java source code generated from the visual model or textual program in Figure 12 or 13.



Property	Value
EMF	
Crossover Policy	HalfAndHalf
Mutation Rate	5.2
Partner Selection Policy	Local

Fig. 12: Visual Modeling Environment in BEYONDwork BS Conf. Facility

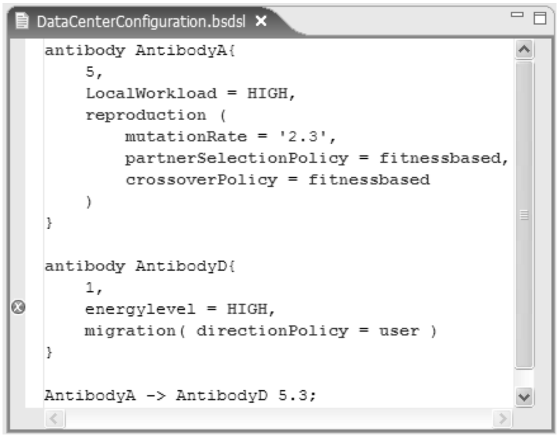
Listing 1.3: An Example Generated Code

```
void setupAntibodiesOfINet(){
  Antibody antibodyA =
   new Antibody( "AntibodyA", 5, LocalWorkload.HIGH,
    new Reproduction(
     2.3, CROSSOVER.FITNESSBASED, PARTNER.FITNESSBASED ) );
  Antibody antibodyD =
   new Antibody( "AntibodyD", 1, EnergyLevel.HIGH,
    new Migration( DirectionPolicy.USER ) );

  ImmuneNetwork inet = getImmuneNetwork();
  inet.add( antibodyA );
  inet.add( antibobyD );
  antibodyA.addAffinity( antibodyD, 5.3 );
}
```

Without the BS configuration facility, agent designers need to know the details on how to implement agents in Java (e.g., how to define new agents, where to implement antibody configuration code, and which iNet API to use.) For example, agent designers need to define a new class extending the `Agent` class provided by a simulator in BEYOND. Also, as the above code fragment shows, they need to write the `setupAntibodiesOfINet()` method using iNet API in order to configure the agent's antibodies. The visual and textual DSLs hide these implementation details and allow agent designers to focus on the design of antibody configurations. In addition, compared with the Java code shown above, a model or program in the BS configuration facility is easier to read and understand.

Figure 14 shows a metamodel of BS configuration model which is partly generated from an environment configuration model in Figure 6. The metamodel consists of five generic metaclasses, i.e., `AntibodyNetwork`, `Antibody`, `Affinity`, `EnvironmentCondition`



```
antibody AntibodyA{
    5,
    LocalWorkload = HIGH,
    reproduction (
        mutationRate = '2.3',
        partnerSelectionPolicy = fitnessbased,
        crossoverPolicy = fitnessbased
    )
}

antibody AntibodyD{
    1,
    energylevel = HIGH,
    migration( directionPolicy = user )
}

AntibodyA -> AntibodyD 5.3;
```

Fig. 13: Textual Programming Environment in BEYONDwork BS Conf. Facility

and `Behavior`, and four metaclasses for certain behaviors, i.e., `Reproduction`, `Replication`, `Migration` and `Death`.

`AntibodyNetwork` represents an antibody network, and its instance is an behavior policy itself, e.g., Figure 12. `Antibody` represents an antibody, and the `name` and `initialConcentration` attributes are its name and the initial concentration value respectively. `Affinity` represents an affinity between `Antibodys`, i.e., the direction and the affinity value. `EnvironmentCondition` represents an environment condition when an antibody invoking its behavior. As described in Section 3.1, the envionment configuration facility defines an envronment configuration model; what environment conditions exist, e.g., `LocalWorkload`, and what categories each environment condition has, e.g., `HEAVY` and `LIGHT`. According to an environment configuration model, two enumeration types, i.e., `Environment` and `Category` are automatically generated through the notion of MSMR. These two enumeration types are used to configure the visual modeling environment (Figure 12), and allows agent designers to configure antibodys' environment conditions.

`Behavior` is a parent metaclass of all agent behaviors. In order to introduce a new agent behavior, administrators define a metaclass representing the new behavior by inheriting the `Behavior` metaclass. Addition/Customization of new behaviors is performed through a MSMR from an agent behavior model to a metamodel in the BS configuration facility. `Reproduction`, `Replication`, `Migration` and `Death` metaclasses are automaitcally generated from an agent behavior model in Section 3.2. Each represents reproduction, replication, migration and death behaviors respectively (see Section 2.1). `Reproduction` invokes a reproduction behavior with its `mutationRate`, `partnerSelectionPolicy` and `crossoverPolicy` attributes. `Replication` invokes a replication behavior. `Migration` invokes a migration behavior with its `directionPolicy` attribute. `Death` invokes a death behavior.

The visual modeling and textual programming facilities are implemented on GMF and oAW respectively. The transformations from visual model/textual programs to Java

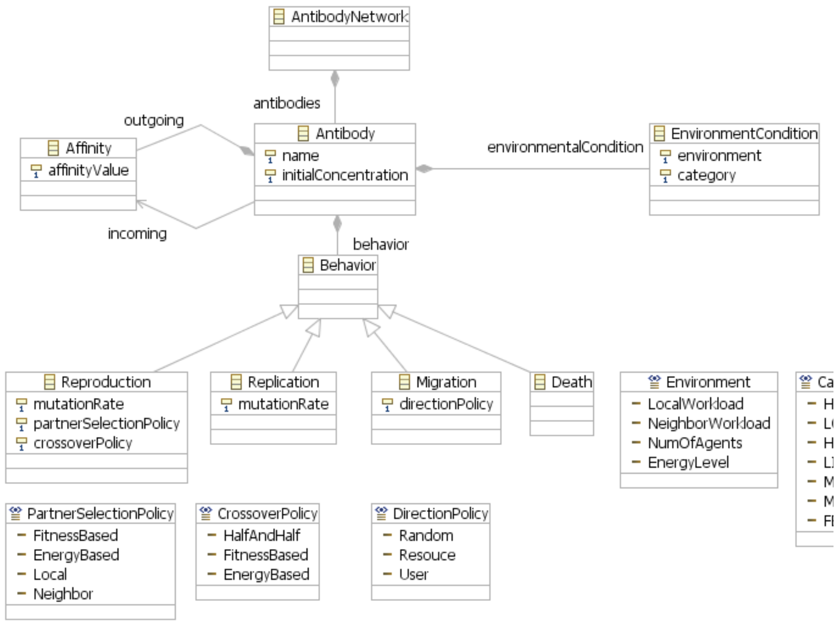


Fig. 14: Metamodel of BS Model

source code are implemented with a model-code transformation engine in oAW as well as the environment configuration facility. MSMR, a model transformation from an environment configuration model and an agent behavior model to a metamodel in the BS configuration facility, is implemented with a model-model transformation engine in oAW.

The BS configuration facility allows agent designers to not only configure an antibody configuration (behavior policy) from scratch, but also investigate and fine tune existing antibody configurations in running agents. In iNet, antibody configurations are evolved automatically via genetic operations (see Section 2.2.2). The BS configuration facility helps agent designers to understand evolved antibody configurations by showing it in a visual manner, and experienced agent designers can fine-tune them by hand.

4 Related Work

Several researches have investigated model-driven development techniques for autonomic computing based on UML, i.e., a general-purpose modeling language [6–9]. Their model tends to be complicated and not easy for application administrators to use. [10] proposes an XML-based language, called Autonomic Computing Policy Language (ACPL), to describe policies for autonomic computing. ACPL is designed as a general-purpose policy language. For example, it provides `Condition` and `Action` tags to describe a condition and an action to take. It can describe any types of policies, but not specialized to certain mechanisms. As well, [11] allows describing pairs of an environment condition and an action through the use of general-purpose textual policy language. The proposed visual DSLs make it easy to understand, define and maintain policies (i.e., agent behavior policies) in autonomic applications rather than general-purpose policy languages.

There are several DSLs to model biological systems such as biochemical networks for simulating and understanding biological systems (e.g., [12, 13]). However, the objective of the DSLs in BEYONDwork is different from theirs; DSLs in BEYONDwork aim to model biological (immunological) mechanisms for building autonomous and adaptive network applications. This work is the first attempt to investigate a DSL for biologically-inspired networking.

[14] provides a DSL, called J2EEML, to configure QoS aspects of Enterprise Java Beans such as response time and message scheduling algorithms in a visual manner. It assumes a stable domain-specific metamodel, and do not address the issue of customization of DSLs, i.e., do not provide means to customize metamodels. BEYONDware allows application administrators not only to use DSLs to model policies, but also to customize DSLs through the notion of Multi-Staged Model Reflection (MSMR). This mechanism allows even application administrators to customize DSLs to reflect the changes in the semantics of domain concepts.

[15] proposes a technique, called *Metamodel Composition*, to extend domain-specific metamodels by reusing and combining existing domain concepts through inheritance. It aims to provide a set of operators to extend domain-specific metamdoels without changing existing metamodels in order to avoid invalidating existing models that comforms to the existing metamodels. The objective of MSMR is different from metamodel composition; MSMR aims to customize domain-specific metamodels through the use of domain-specific models. BEYONDwork currently changes a domain-specific metamodel directly and does not consider the reuse of existing metamodels, however, metamodel composition can be used as a strategy to customize a domain-specific metamodel in BEYONDwork.

A model transformation from a lower-level (e.g., model) to a higher-level (e.g., metamodel) is called *promotion*. [16] leverages this technique to create a new domain-specific metamodel from a model, but they uses a general-purpose modeling language to describe a model that to be *promoted* to a metamodel. MSMR also leverages promotion, but uses a DSL to customize other DSLs. It simplifies the customization of DSLs and allows even application administrators to customize DSLs.

5 Conclusion

BEYONDwork is an MDD environment to reduce the complexity for designing and maintaining the operational policies in autonomic network applications. It provides (1) visual and textual DSLs to graphically model the operational policies in biologically-inspired autonomic network applications, and (2) a supporting tool that validates the operational policies defined in the proposed DSLs and transform them to program code. BEYONDwork enables rapid and intuitive configuration of operational policies. In addition, BEYONDwork addresses an issue in DSLs: low flexibility for customization. Through the notion of Multi-Staged Model Reflection (MSMR), it allows users (even non-programmers) to graphically customize DSL's grammar (or metamodel).

References

1. G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, and B. Selic, “An mda manifesto,” in *The MDA journal: Model Driven Architecture Straight from the Masters*. Meghan-Kiffer Press, December 2004, ch. 11.
2. G. Cook, “Domain-specific modeling and model-driven architecture,” in *The MDA journal: Model Driven Architecture Straight from the Masters*. Meghan-Kiffer Press, December 2004, ch. 5.
3. C. Lee, H. Wada, and J. Suzuki, “Towards a biologically-inspired architecture for self-regulatory and evolvable network applications,” in *Advances in Biologically Inspired Information Systems Models, Methods, and Tools*. Springer, July 2007.
4. N. K. Jerne, “Idiotypic networks and other preconceived ideas,” *Immunological Review*, 1984.
5. C. Berek, “Somatic hypermutation and b-cell receptor selection as regulators of the immune response,” *Transfusion Medicine and Hemotherapy*, 2005.
6. I. Trencansky, R. Cervenka, and D. Greenwood, “Applying a uml-based agent modeling language to the autonomic computing domain,” *ACM OOPSLA onward! track*, October 2006.
7. J. Peña, M. Hinchey, R. Sterritt, A. Cortés, and M. Resinas, “A model-driven architecture approach for modeling, specifying and deploying policies in autonomous and autonomic systems,” *Int’l Symposium on Dependable Autonomic and Secure Computing*, October 2006.
8. B. B. H. Kasinger, “Towards a model-driven software engineering methodology for organic computing systems,” *Int’l Conference on Computational Intelligence*, July 2001.
9. M. Rohr, M. Boskovic, S. Giesecke, and W. Hasselbring, “Model-driven development of self-managing software systems,” *ACM/IEEE MoDELS Workshop on Models@Runtime*, October 2006.
10. D. Agrawal, K.-W. Lee, and J. Lobo, “Policy-based management of networked computing systems,” *IEEE Communications Magazine*, October 2005.
11. J. Dubus and P. Merle, “Applying omg d&c specification and eca rules for autonomous distributed component-based systems,” *ACM/IEEE MoDELS Workshop on Models@Runtime*, October 2006.
12. M. Hucka, A. Finney, B. Bornstein, S. Keating, B. Shapiro, J. Matthews, B. Kovitz, M. Schilstra, A. Funahashi, J. Doyle, and H. Kitano, “Evolving a lingua franca and associated software infrastructure for computational systems biology: The systems biology markup language (sbml) project,” *Systems Biology Journal*, June 2004.
13. F. Kolpakov, “Biouml - framework for visual modeling and simulation biological systems,” in *Int’l Conference Bioinformatics of Genome Regulation and Structure*, July 2002.
14. J. White, D. Schmidt, and A. Gokhale, “Simplifying autonomic enterprise java bean applications via model-driven engineering and simulation,” *The Journal of Software and System Modeling*, 2007, to appear.
15. A. Ledeczi, G. Nordstrom, G. Karsai, P. Volgyesi, and M. Maroti, “On metamodel composition,” *IEEE Int’l Conference on Control Applications*, September 2001.
16. F. Jouault and J. Bézivin, “Km3: a dsl for metamodel specification,” *IFIP Int’l Conference on Formal Methods for Open Object-Based Distributed Systems*, October 2006.