5
Gerando Código a Partir de Diagramas de MAS-ML

A fim de implementar um modelo de sistema que usa a linguagem de modelagem MAS-ML, é necessário refinar os modelos e ser capaz de gerar código. Os modelos descritos no nível de abstração do agente precisam ser transformados em código orientado a objetos. Um transformador chamado MAS-ML2Java foi criado a fim de gerar código a partir de modelos de MAS-ML. O processo de transformação baseia-se em uma arquitetura[4] abstrata orientada a objetos para SMAs gerados a partir do metamodelo de MAS-ML. Essa arquitetura define um conjunto de módulos de componentes orientados a objetos e independentes do domínio e seus relacionamentos. Cada abstração, propriedade e relacionamento definidos no metamodelo de MAS-ML é representado na arquitetura abstrata.

O processo de transformação (Silva et al., 2004d) usa um conjunto de regras de transformação independentes do domínio e um conjunto de regras de transformação dependentes do domínio. As regras de transformação independentes do domínio (ou regras básicas) geram os módulos de componentes representados na arquitetura abstrata. As regras dependentes do domínio geram os componentes originados a partir dos modelos de MAS-ML relacionados ao domínio da aplicação. Os componentes dependentes do domínio estendem os componentes definidos na arquitetura abstrata. O conjunto de componentes gerados orientados a objetos representa as entidades da aplicação, suas propriedades e relacionamentos conforme modelado nos diagramas estruturais de MAS-ML.

O transformador MAS-ML2Java foi desenvolvido usando TXL (TXL, 2004), uma linguagem de programação desenvolvida para oferecer suporte à programação transformacional. Todas as entidades, propriedades e

relacionamentos representados nos diagramas estruturais de MAS-ML são mapeados para classes, atributos e métodos. Como os diagramas estruturais não descrevem as informações sobre a execução do sistema, representadas no diagrama dinâmico, o código gerado pelo transformador não é executável sem a adição de código Java extra pelo implementador (como é o caso nas ferramentas de suporte de UML padrão (Together, 2004; IBM, 2004)).

5.1. A Linguagem de Programação TXL

Basicamente, TXL transforma um texto de entrada, descrito de acordo com uma gramática especificada, em um texto de saída, gerado de acordo com a gramática que usa um conjunto de regras de transformação. Para transformar modelos estruturais de MAS-ML em Java, foi necessário criar uma gramática a fim de definir a entrada (modelos de MAS-ML) e um conjunto de regras para transformar essa entrada em classes Java. TXL foi usada porque é uma linguagem especificamente desenvolvida para tarefas de transformação de fonte. Ela também é conhecida como uma boa linguagem para o desenvolvimento da extração de arquitetura e recuperação a partir da fonte (TXL, 2004).

A unidade básica de uma gramática de TXL é a declaração *define*. Cada declaração *define* fornece um conjunto ordenado de formas alternativas especificadas como uma seqüência de símbolos terminais e não-terminais. A barra vertical, '|', é usada para separar alternativas. Os símbolos não-terminais são colocados entre colchetes '[ ]', os demais símbolos são terminais. Alguns símbolos não-terminais são predefinidos em TXL: [id] representa qualquer identificador que comece com uma letra, [NL] força uma nova linha de saída, [IN] endenta as linhas de saída seguintes, [EX] "exdenta" todas as linhas de saídas seguintes e outros. O tipo [program] não-terminal é o símbolo de objetivo para todos os programas TXL (o tipo atribuído a todas as entradas para fins de interpretação - *parsing*). Os programas TXL devem conter uma definição para *program*.

[4] Uma arquitetura de software define a estrutura dos componentes de um programa/sistema, seus inter-relacionamentos, princípios e diretrizes que governam seu design e sua evolução ao longo do tempo (Garlan et al., 1995).

A transformação aplicada à árvore de entrada analisada de acordo com a gramática é especificada como um conjunto de *regras* e *funções* de transformação. Cada função ou regra TXL é composta por um *nome* que é um identificador, um *tipo* que é o tipo não terminal que será transformado, um *padrão* ao qual a árvore de argumento de funções deve corresponder para que a função o transforme, e uma *substituição* que é o resultado da função quando a árvore é correspondente. Se a árvore de argumento for correspondente ao padrão, o resultado é a substituição, caso contrário, o resultado é a árvore de argumento (inalterada). Cada programa TXL deve ter uma regra ou função chamada *main*. A execução de um programa TXL consiste na aplicação dessa regra a toda árvore de entrada analisada gramaticalmente. Se outras regras tiverem de ser aplicadas, então a regra principal deve ser chamada de forma explícita.

5.2. Transformador MAS-ML2Java

O processo definido para transformar os modelos da linguagem MAS-ML em Java é composto por duas fases: transformação de entidades e transformação de relacionamentos. A fase de transformação de entidades transforma as entidades e suas propriedades, conforme descrito nos modelos de MAS-ML, em código Java. Essa fase não trata dos relacionamentos. A transformação de relacionamentos em código Java descrita nos modelos de MAS-ML ocorre na fase de transformação de relacionamentos. Essas duas fases de transformação usam duas gramáticas semelhantes e dois conjuntos de regras diferentes. Cada fase de MAS-ML2Java é composta pelos arquivos a seguir: a entrada, o programa TXL principal (.txl), a gramática (.grammar), o conjunto de regras (.rule) e o resultado gerado. O programa TXL principal interpreta a entrada de acordo com a gramática e chama o conjunto de regras que geram os resultados.

5.2.1. Fase de Transformação de Entidades

A entrada para essa fase descreve as entidades, suas propriedades e os relacionamentos entre as entidades de uma aplicação. O resultado é um arquivo que contém o conjunto de relacionamentos descrito na entrada e o código gerado associado às entidades e suas propriedades. A gramática usada nessa fase, chamada de gramática de MAS-ML, foi definida com base no metamodelo de

MAS-ML. Ela define a estrutura das entidades, propriedades e relacionamentos descritos na entrada e a estrutura dos relacionamentos e classes gerados nos resultados. Para transformar a entrada em resultados, usamos dois conjuntos de regras: regras para entidades independentes do domínio e regras de entidades dependentes do domínio, conforme ilustrado na Figura 33. As regras para entidades independentes do domínio geram um conjunto de classes que formam a base de qualquer aplicação de SMAs, junto com os relacionamentos independentes do domínio. O conjunto de classes básicas caracteriza a arquitetura abstrata orientada a objetos descrita na Seção 5.4.1. As regras para entidades dependentes do domínio criam um conjunto de classes gerado de acordo com o domínio da aplicação.

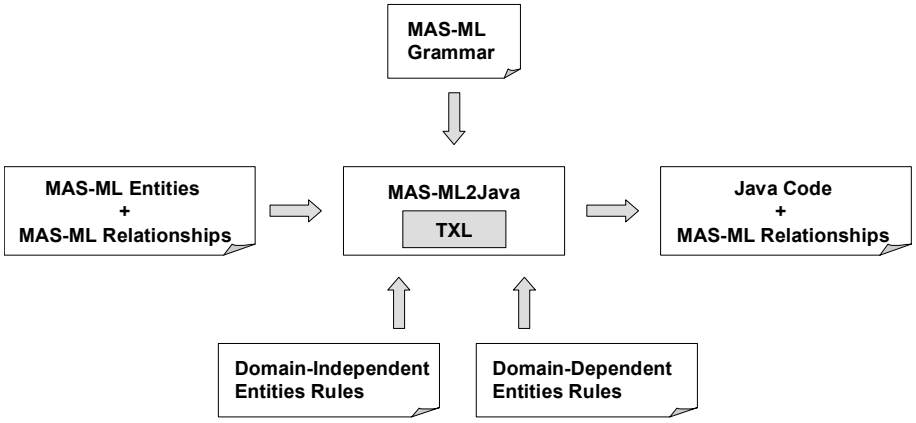

Figura 33 – Transformação das entidades.

5.2.2. Fase de Transformação de Relacionamentos

O resultado da fase anterior é a entrada da fase atual, ou seja, a entrada dessa fase contém o conjunto de relacionamentos da linguagem MAS-ML não transformado na fase anterior e o código gerado associado às entidades e suas propriedades. O resultado dessa fase é um conjunto de arquivos de Java cada um contendo uma classe Java. A gramática usada nessa fase é um subconjunto da gramática de MAS-ML definida na fase anterior, uma vez que não define as propriedades e as entidades de MAS-ML. As propriedades e entidades de MAS-ML já foram transformadas em classes e atributos. A gramática define a estrutura dos relacionamentos e classes descritos na entrada e a estrutura do conjunto de classes do resultado. O conjunto de regras dessa fase são regras de relacionamentos dependentes do domínio, conforme ilustrado na Figura 34. Essas

regras modificam o código gerado na fase anterior de acordo com os relacionamentos definidos na aplicação.

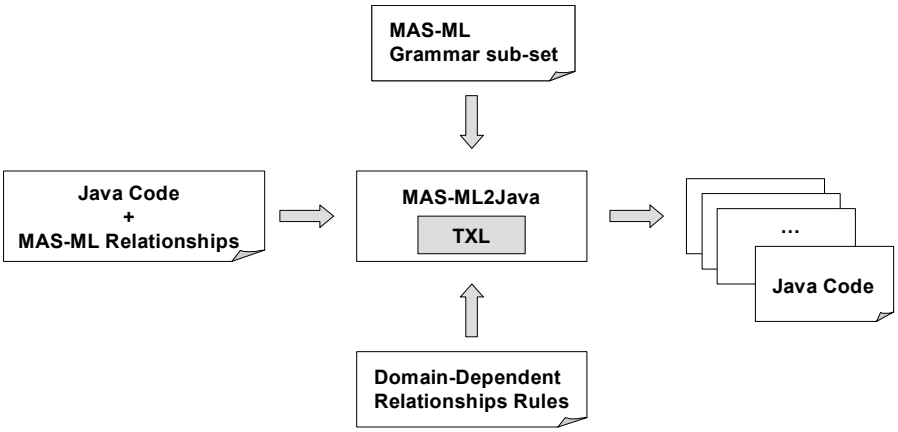

Figura 34 – Transformação de relacionamentos.

5.3.
A Gramática de MAS-ML

A gramática de MAS-ML usada pelo transformador foi gerada a partir da descrição do metamodelo de MAS-ML. Usando essa gramática, é possível descrever todas as informações descritas nos diagramas estruturais de MAS-ML. É possível descrever as entidades, suas propriedades e relacionamentos.

No Capítulo 3, os SMAs são descritos como uma composição de ambientes, organizações, agentes, objetos, papéis de objeto, papéis do agente e seus relacionamentos. Além do mais, os SMAs são compostos por um ambiente, uma organização principal, pelo menos um agente, um papel do agente (exercido pelo agente) e o conjunto de relacionamentos que ligam as entidades. A gramática de MAS-ML usada na fase de transformação de entidades é definida com base nessas premissas. Ela possibilita descrever todas as características representadas nos três diagramas estruturais (diagramas de classes, organização e papel).

A gramática usada na fase de transformação de relacionamentos é um subconjunto da gramática de MAS-ML usada na fase de transformação de entidades. A gramática usada nessa segunda fase não descreve ambientes, organizações, agentes, papéis de objeto e papéis do agente porque essas entidades foram transformadas em classes Java durante a primeira fase. Portanto, ela descreve apenas classes e relacionamentos. Como essa gramática é um subconjunto da gramática usada na primeira fase, ela não será detalhada nesta seção. As duas gramáticas são apresentadas em detalhes no Apêndice I.

A gramática de MAS-ML é iniciada definindo o não-terminal *program*, que é a principal entidade de todas as gramáticas. *Program* é composto por *relationshipDescription*, *basicEntities* e *otherEntities*, que são não-terminais. O não-terminal *basicEntities* é composto pelos não-terminais *environment*, *organization*, *agentRole* e *agent*. Cada não-terminal define uma classe de entidade específica que deve existir nos SMAs. Conforme mencionado anteriormente, os SMAs são compostos por um ambiente, uma organização principal e pelo menos um agente e um papel do agente. Portanto, a classe de entidade dessas entidades deve ser descrita em todos os SMAs.

O não-terminal *otherEntities* define outras classes de entidade, como classe e classe de papel de objeto, e também outras classes de ambiente, classes de organização, classes de agente e classes de papel de agente que também podem compor o SMA. O não-terminal *relationshipDescription* define os tipos de relacionamentos que podem ser usados para ligar entidades definidas no MAS: relacionamentos *ownership*, *inhabit*, *play*, *control*, *dependency*, *association*, *aggregation* e *specialization*. A Figura 35 mostra a parte da gramática de MAS-ML que define *relationshipDescription*, *basicEntities* e *otherEntities*.

```
define program
      [relationshipDescription+][basicEntities][otherEntities]
end define
define basicEntities
       [environment] [NL][NL]
       [organization] [NL][NL]
       [agentRole] [NL][NL]
       [agent] [NL][NL]
end define
define otherEntities
      [entity*]
end define
define entity
       [environment] [NL][NL]
      | [organization] [NL][NL]
      | [agentRole] [NL][NL]
      | [agent] [NL][NL]
      | [objectRole] [NL][NL]
      | [entityClass] [NL][NL]
end defin-{}-e
define relationshipDescription
      'RELATIONSHIP [relationship][NL] [NL]
end define
define relationship
      [ownership]
      | [inhabit]
      | [play]
      | [control]
      | [dependency]
      | [association]
```

```
      | [aggregation]
      | [specialization]
end define
```

Figura 35 –A gramática de MAS-ML usada na fase de transformação de entidades(parcial)

Para exemplificar a definição de um relacionamento e de uma classe de entidade apresentada na gramática de MAS-ML, o relacionamento *play* e a *agent class* são apresentados nesta seção. O relacionamento *play* (vide Figura 36) é composto pela identificação da entidade que exercerá o papel (agente, organização ou objeto), a multiplicidade associada à entidade (sempre uma), a palavra reservada PLAYS, a multiplicidade do papel que será exercido, a identificação do papel (papel do agente ou papel de objeto), a palavra reservada IN e a identificação da organização em que o papel será exercido. Usando essa estrutura, é possível descrever a entidade que exercerá o papel, o papel que será exercido, a organização em que o papel será exercido e o número de instâncias de papel que podem ser exercidas pela entidade. O número de instâncias de entidade que exercem a instância de papel é sempre um, pois uma instância de papel não pode ser exercida por mais de uma entidade. Cada instância de papel é exercida por apenas uma instância de entidade.

```
define play
      [agentID] '1 'PLAYS [multiplicity][agentRoleID] 'IN
            [organizationID]
      | [organizationID] '1 'PLAYS [multiplicity][agentRoleID] 'IN
            [organizationID]
      | [entityClassID] '1 'PLAYS [multiplicity][objectRoleID] 'IN
            [organizationID]
end define
```

Figura 36 – A gramática do relacionamento *play*.

Uma classe do agente é descrita com base em sua identificação, objetivos, crenças, planos e ações (consulte Figura 37). Ela define as crenças, objetivos, ações e planos iniciais das instâncias do agente. Observe que uma classe do agente pode definir zero ou mais objetivos, crenças, planos e ações. Como ela é transformada em uma classe, uma classe do agente também é definida como uma classe. A transformação de um agente de aplicação em uma classe é explicada na Seção 5.4.2.1.

```
define agent
      'AGENT [NL] '(
            [agentID][IN][NL]
            [goal*][NL]
            [belief*][NL]
            [plan*][NL]
```

```
            [actionPrePostCondition*][NL][EX]
      ')
      |[entityClass*]
      %an agent class is transformed into an OO class
end define
```

Figura 37 – A gramática da classe do agente.

Um objetivo é definido como um objetivo simples ou um objetivo composto. Conforme apresentado na Seção 4.2.1, um objetivo é definido como um atributo, e todos os objetivos são associados a um plano que alcança o objetivo. Uma crença também é descrita como um atributo definido por um tipo, um nome e um valor. As gramáticas de crença e objetivo são apresentadas na Figura 38.

```
define goal       % Goal is an attribute
      [goalLeaf]
      | [goalComposite]
end define
define goalLeaf
      'GOAL '( [attrBody] [goalPlans+] ')
      ...
end define
define goalComposite
      'GOAL '( [attrBody] [goalPlans*] 'SUBGOAL '( [goal+] ')')
      ...
end define
define goalPlans
      'RELATED 'TO 'PLAN [stringlit]
      ...
end define
define belief     % Beleif is an attribute
      'BELIEF '( [attrBody] ')
      ...
end define
define attrBody
      [varType] ': [varName] '= [varValue]
end define
```

Figura 38 – As gramáticas de objetivo e crença.

Um plano é identificado por seu nome, seu conjunto de ações associadas ao plano e o objetivo que o plano alcança (vide Figura 39). Ao descrever um plano, o conjunto de ações e o objetivo associado ao plano podem ser omitidos. O objetivo e as ações podem ser associados ao plano, quando ele é instanciado.

```
define plan
      'PLAN '( [planName] [planActions*] [planGoal]')
      |[entityClass*]
      ...
end define
define planActions
      'COMPOSED 'OF 'ACTION [stringlit]
      ...
end define
define planGoal
```

```
      'RELATED 'TO 'GOAL [varName]
      ...
end define
```

Figura 39 – A gramática de plano.

Uma ação é definida por seu nome e suas precondições e pós-condições (consulte Figura 40). As precondições e as pós-condições também são definidas como atributos. Observe que, como planos e ações são transformadas em classes, um plano e uma ação também são definidos como uma classe. A Seção 5.4.2.1 detalha essas transformações.

```
define actionPrePostCondition
      [action][NL][preCondition*][postCondition*]
      |[entityClass*]
end define
define action
      'ACTION [actionAssignment]
      ...
end define
define preCondition
      'PRECONDITION '( [attrBody] ')
      ...
end define
define postCondition
      'POSTCONDITION '( [attrBody] ')
      ...
end define
```

Figura 40 – A gramática de ação.

Na Figura 41,há um exemplo parcial de uma entrada de MAS-ML criada de acordo com a definição do relacionamento *play* e a classe do agente. No exemplo, o relacionamento *play* define que uma instância *User_Agent* pode exercer zero ou mais papéis *Buyer* em uma organização chamada *General_Store*. Também define que uma instância de papel *Buyer* é exercida por uma instância *User_Agent*. Em seguida, a classe *User_Agent* é definida como tendo um objetivo, crença, plano e ação. O Apêndice II apresenta um exemplo completo descrito usando a gramática de MAS-ML.

```
RELATIONSHIP User_Agent 1 PLAYS  0..* Buyer IN General_Store

AGENT
( User_Agent
      GOAL ( "Boolean" :  "itemBought" = "true"
            RELATED TO PLAN "Searching_seller"
      )
      BELIEF ("Money" : "ammountOfMoney" = "$1000.00")
      PLAN ( Searching_seller
                  COMPOSED OF ACTION "Search_for_seller"
                  RELATED TO GOAL "itemBought"
      )
      ACTION Search_for_seller
            POSTCONDITION ( "Boolean" : "seller" = "true")
```

```
)
```

Figura 41 – Descrição de um relacionamento *play* e de um agente usando a gramática.

5.4. Regras de Transformação da Linguagem MAS-ML

As regras de transformação de MAS-ML são divididas em três grupos de regras: regras para entidades independentes do domínio, regras para entidades dependentes do domínio e regras para relacionamentos dependentes do domínio. Os relacionamentos independentes do domínio são gerados pelas regras de entidades independentes do domínio.

5.4.1. Regras de Transformação para Entidades Independentes do Domínio

A fim de transformar todas as entidades e suas propriedades apresentadas nos diagramas estruturais de MAS-ML em código Java, precisamos criar classes, atributos e métodos para representá-las. Entretanto, as classes de entidade, como classes de papel de agente, agente, organização e ambiente, normalmente usadas em SMAs, não são abstrações disponíveis em sistemas OO. Não é possível mapear diretamente essas classes de entidade em classes OO, porque têm características comportamentais e estruturais diferentes. Por exemplo, um agente possui objetivos, crenças, planos e ações, e uma classe tem atributos e métodos. A única entidade usada em MAS-ML que pode ser diretamente mapeada para o código JAVA é a classe. A definição da classe usada em diagramas de MAS-ML e a definição de classe apresentada em Java compartilham as mesmas características comportamentais e estruturais: atributos e métodos.

A fim de implementar entidades de SMAs usando uma linguagem de programação OO como Java, é necessário criar conjuntos de classes que representam as entidades que não podem ser diretamente mapeadas. Cada conjunto de classes representa a estrutura de uma entidade de SMAs e suas propriedades associadas. Cada conjunto de classes é composto por uma classe abstrata que representa a entidade e outras classes abstratas e concretas que representam as propriedades da entidade. Por exemplo, em cada implementação de SMAs, há um conjunto de classes que representam a entidade abstrata *agent* e suas propriedades. Os agentes concretos definidos em uma aplicação estendem o agente abstrato e herdam todas as suas características, ou seja, herdam as

características comuns definidas para cada agente. As regras que criam esses conjuntos de classes são chamadas regras para entidades independentes do domínio, porque, independente da aplicação, o conjunto de classes geradas é exatamente o mesmo. O conjunto de classes, atributos e métodos gerados pelas regras para entidades independentes do domínio compõe a arquitetura abstrata OO para SMAs.

As regras para entidades independentes do domínio são: a regra de agentes básica, a regra de organizações básica, a regra de papéis básica, a regra de papéis de objeto básica e a regra de ambientes básica. Ao agrupar todos os conjuntos de classes geradas para cada entidade de SMAs, a arquitetura abstrata OO para SMAs é criada.

5.4.1.1. A Regra de Agentes Básica

A regra de agentes básica é responsável pela criação do conjunto de classes que representa um agente abstrato e suas propriedades relacionadas. No código OO, o conjunto de classes que representa um agente é descrito com base na definição de um agente apresentado em MAS-ML. A seguir, resumimos a definição de um agente descrevendo suas propriedades e seus relacionamentos básicos (independentes do domínio) que influenciam a definição do conjunto de classes:

1. um agente possui crenças, objetivos, planos e ações;
2. um agente possui uma identificação;
3. agentes interagem enviando e recebendo mensagens enquanto exercem papéis;
4. os agentes residem em apenas um ambiente;
5. os agentes exercem pelo menos um papel em uma organização;
6. os agentes podem exercer diferentes papéis nas organizações.

A entidade *agent* de MAS-ML é representada por uma classe abstrata chamada *Agent* e um conjunto de classes associado ao agente que representa suas propriedades. Como todos os agentes têm um conjunto de objetivos, crenças, planos e ações, a classe abstrata que representará um agente deve definir os atributos para armazenar os valores associados a essas propriedades. Portanto, é

necessário definir como os objetivos, as crenças, os planos e as ações serão representados no código OO.

Os objetivos e as crenças são definidos como atributos. Um objetivo ou uma crença pode ser de qualquer tipo, pode ter qualquer nome e valor. Além disso, podem ser criados, modificados e excluídos durante a execução de um agente. Uma classe chamada *Belief* é criada para representar as crenças de um agente. Ela define três atributos: o tipo da crença, o nome da crença e o valor da crença. As crenças de um agente são instâncias da classe *Belief*. Uma crença pode ser modificada alterando um de seus atributos e também pode ser excluída destruindo a instância da crença.

Para representar os objetivos de um agente, o padrão Composite (Gamma et al., 1995) é usado porque os objetivos podem ter subobjetivos. Um objetivo pode ser um objetivo composto (um objetivo composto por outros objetivos) representado pela classe *CompositeGoal* ou um objetivo simples representado pela classe *LeafGoal*. A estrutura básica dos objetivos descritos pela classe abstrata chamada *Goal* é definida usando cinco atributos. Um atributo armazena a lista de planos associados ao objetivo, outro armazena o estado do objetivo (alcançado ou não), e três outros atributos armazenam o tipo, o nome e o valor do objetivo. A classe *CompositeGoal* e *LeafGoal* são especializações da classe *Goal*. Um objetivo de um agente é uma instância da classe *CompositeGoal* ou *LeafGoal*.

Um plano é composto por um conjunto de ações e está relacionado a um objetivo que pode ser alcançado. Cada plano define a seqüência (ou a ordem) das ações que serão exercidas quando o plano é chamado. Como cada plano define sua execução particular, não é possível criar uma classe concreta chamada *Plan* para representar qualquer plano. Uma classe abstrata chamada *Plan* é criada para definir a estrutura abstrata dos planos. Ela define dois atributos e um método. Um atributo armazena o objetivo, e o outro, uma lista de ações. O método chamado `execute` define uma execução básica dos planos dando início às ações na lista de acordo com a seqüência. A execução básica pode ser especializada (re-implementada) pelos planos concretos (planos definidos na aplicação) que estendem a classe abstrata *Plan*.

As ações definem as tarefas dos agentes. Uma ação pode ter precondições e pós-condições. Como cada ação define diferentes execuções, uma classe abstrata chamada *Action* é criada para representar a estrutura das ações. Essa classe define

dois atributos: as precondições e as pós-condições e um método chamado `execute` que devem ser implementados pelas ações concretas definidas na aplicação. Como as condições são definidas como atributos, é definida uma classe chamada *Condition* que tenha o tipo de condição, o nome da condição e o valor da condição dos atributos.

A classe abstrata Agent define onze atributos. Cinco atributos armazenam os valores associados às propriedades do agente: listas de objetivos simples e compostos, a lista de crenças, a lista de planos e a lista de ações. Três outros atributos estão relacionados aos relacionamentos básicos entre agentes. Um agente também deve armazenar o ambiente em que reside, a lista de papéis que está exercendo e as organizações em que está exercendo papéis. Os três últimos atributos armazenam o nome do agente e as mensagens enviadas e recebidas pelo agente. A Figura 42 ilustra a classe abstrata Agent e as classes associadas a ela.

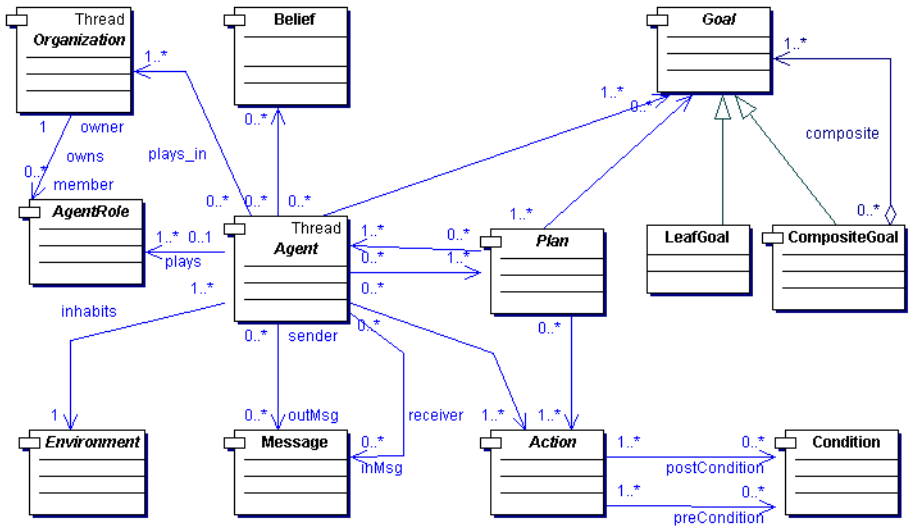

Figura 42 – A classe abstrata Agent, as classes que representam suas propriedades e outras classes relacionadas a ela.

5.4.1.2. A Regra de Organizações Básica

A regra de organizações básica é responsável pela criação do conjunto de classes que representam uma organização e suas propriedades relacionadas. Resumimos a seguir a definição de uma organização de acordo com MAS-ML, apresentando suas propriedades e seus relacionamentos básicos (independentes do domínio):

1. uma organização possui crenças, objetivos, planos, ações e axiomas;

2. uma organização possui uma identificação;
3. as organizações interagem enviando e recebendo mensagens;
4. as organizações residem em apenas um ambiente;
5. as organizações definem os papéis do agente e os papéis de objeto;
6. as organizações definem suborganizações;
7. as suborganizações exercem pelo menos um papel em uma organização;
8. as suborganizações podem exercer diferentes papéis nas organizações.

Em um código OO, a entidade de MAS-ML chamada de organização é representada por uma classe abstrata chamada *Organization* e um conjunto de classes associado à organização que representa suas propriedades. Semelhante à classe *Agent*, a classe *Organization* também define atributos para armazenar os valores das propriedades de uma instância de organização: a lista de objetivos simples e compostos, a lista de crenças, a lista de planos, a lista de ações e a lista de axiomas. As características dos objetivos, crenças, planos e ações definidas pelas organizações e por agentes são as mesmas. Dessa forma, não serão mais uma vez descritas nesta seção.

Em MAS-ML, um axioma é representado como um atributo. Um axioma tem um tipo, um nome e um valor. Portanto, uma classe chamada *Axiom* é criada para representar os axiomas de uma organização. A classe define três atributos para armazenar o tipo do axioma, o nome do axioma e o valor do mesmo.

A classe *Organization* também define seis outros atributos relacionados aos relacionamentos básicos. Ela define atributos para armazenar as suborganizações de uma instância de organização e os papéis (papéis do agente e objeto) que são membros da instância de organização. Como a classe *Agent*, a classe *Organization* define atributos para armazenar o ambiente em que a instância de organização reside, a lista dos papéis que a instância de organização está exercendo e as organizações em que a instância de organização está exercendo os papéis. Outros três atributos armazenam o nome da organização e as mensagens enviadas e recebidas pela organização. A Figura 43 mostra a classe abstrata *Organization* e as classes associadas a ela.

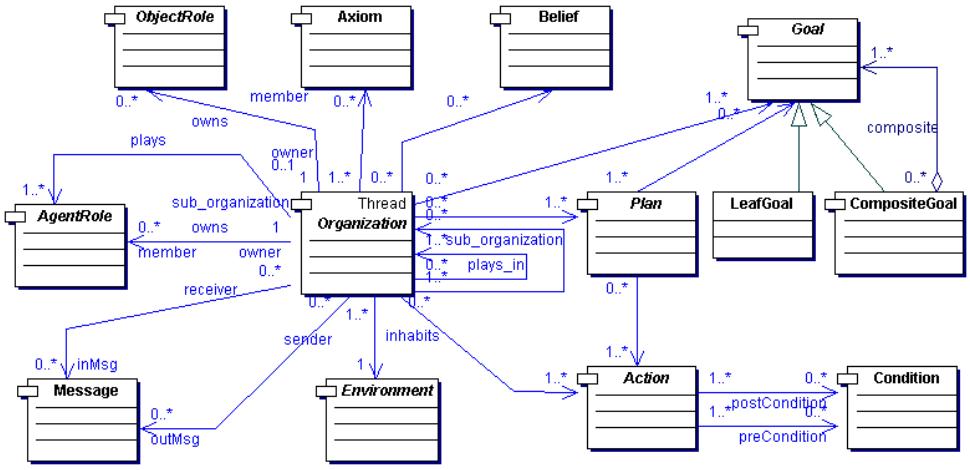

Figura 43 – A classe abstrata Organization, as classes que representam suas propriedades e outras classes relacionadas a ela.

5.4.1.3. A Regra de Papéis de Agentes Básica

A regra de papéis de agentes básica é responsável pela criação do conjunto de classes que representam um papel de agente abstrato e suas propriedades relacionadas. O conjunto de classes que representa o papel de agente e suas propriedades é definido com base na definição de um papel de agente, conforme resumido a seguir:

1. um papel de agente define objetivos, crenças, deveres, direitos e protocolos;
2. um papel de agente possui uma identificação;
3. uma instância de papel de agente é exercida por um agente ou uma organização;
4. cada instância de papel de agente é um membro de uma organização.

A entidade *agent role* de MAS-ML é representada por uma classe abstrata chamada *Agent Role* e um conjunto de classes associado ao papel de agente que representa suas propriedades. Os objetivos e as crenças de um papel de agente são semelhantes aos objetivos e crenças de um agente, e, então, as classes *Goals* e *Beliefs* também serão usadas para representar objetivos e crenças de papéis de agente. Contudo, um objetivo definido por um papel de agente não está relacionado a planos uma vez que papéis de agente não definem planos.

Os deveres e direitos definidos por um papel de agente estão relacionados às ações que o agente, que está exercendo o papel, deve executar e tem permissão

para executar, respectivamente. Portanto, as classes chamadas *Duty* e *Right* identificam apenas um atributo para representar as ações associadas a elas.

Um protocolo define a seqüência (ou a ordem) das mensagens que são enviadas ou recebidas pelo agente que está exercendo o papel. Uma classe abstrata chamada *Protocol* é criada para definir a estrutura abstrata dos protocolos. A classe abstrata *Protocol* define apenas um atributo que é a lista de mensagens e um método abstrato. Os protocolos concretos descritos em uma aplicação estendem a classe *Protocol* e implementam o método que descreve a seqüência de mensagens. Uma mensagem é definida por um rótulo que especifica o tipo de mensagem, por seu conteúdo e pelo emissor e o destinatário. Uma classe chamada *Message* é criada para representar mensagens com essas características.

A classe *Agent Role* define seis atributos para armazenar os valores das propriedades de uma instância de papel do agente: os objetivos compostos e simples, as crenças, os deveres, os direitos e os protocolos. A classe também define atributos para armazenar a organização que a possui e a entidade (agente ou organização) que está exercendo a instância de papel. A Figura 44 ilustra a classe abstrata *Agent Role* e as classes relacionadas a ela.

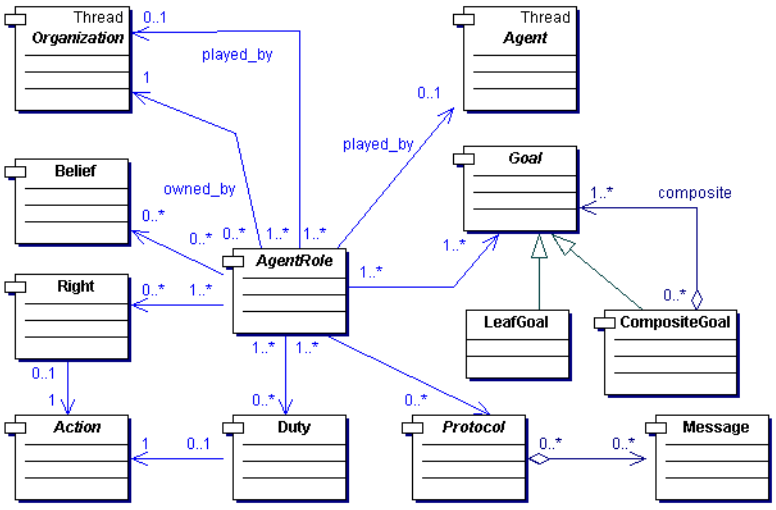

Figura 44 – A classe abstrata Agent Role, as classes relacionadas a suas propriedades e outras classes relacionadas a ela.

5.4.1.4. A Regra de Papéis de Objetos Básica

A regra de papéis de objetos básica é responsável pela criação de uma classe abstrata chamada *Object Role* que representa uma entidade de MAS-ML chamada

object role. A classe é definida com base na definição de um papel de objeto resumida a seguir:

1. um papel de objeto é definido por meio de seus atributos e métodos;
2. um papel de objeto possui um identificador;
3. as instâncias de papel de objeto são exercidas por um objeto;
4. cada instância de papel de objeto é um membro de uma organização.

A classe *Object Role* é criada com três atributos conforme mostrado na Figura 45. Um é o identificador, o segundo representa a organização que possui a instância de papel, e o terceiro representa o objeto que está exercendo a instância de papel. Como papéis de objeto são definidos por meio de atributos e métodos, nenhuma classe foi criada para representar as propriedades de papéis de objeto.

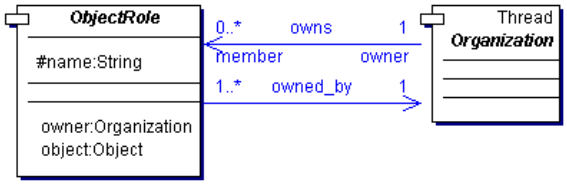

Figura 45 – A classe abstrata Object Role e a classe Organization relacionada a ela.

5.4.1.5. A Regra de Ambientes Básica

A regra de ambientes básica é responsável pela criação de uma classe abstrata chamada *Environment* que representa a entidade de MAS-ML *environment* cuja definição está apresentada a seguir:

1. um ambiente pode ser passivo ou ativo;
2. um ambiente possui um identificador;
3. um ambiente passivo é definido por meio de seus atributos e métodos;
4. um ambiente ativo é definido por meio de objetivos, crenças, planos e ações;
5. em um ambiente residem agentes, organizações e objetos.

A classe *Environment* é criada com quatro atributos conforme ilustrado na Figura 46. Um é o identificador, e os outros três armazenam os agentes, organizações e objetos que residem nele. Como as propriedades de um ambiente dependem de sua definição no domínio da aplicação, não pode ser definido nenhum atributo para representar as propriedades do ambiente usando essa regra independente de domínio.

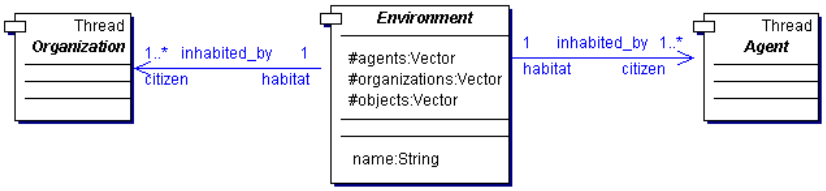

Figura 46 – A classe abstrata Environment e as classes relacionadas a ela.

A Figura 47 mostra a arquitetura abstrata OO para SMAs identificando todas as classes que representam as entidades e suas propriedades e os relacionamentos entre essas classes de acordo com os relacionamentos entre as entidades de MAS-ML. Conforme mencionado anteriormente, a arquitetura abstrata foi gerada com base no metamodelo de MAS-ML.

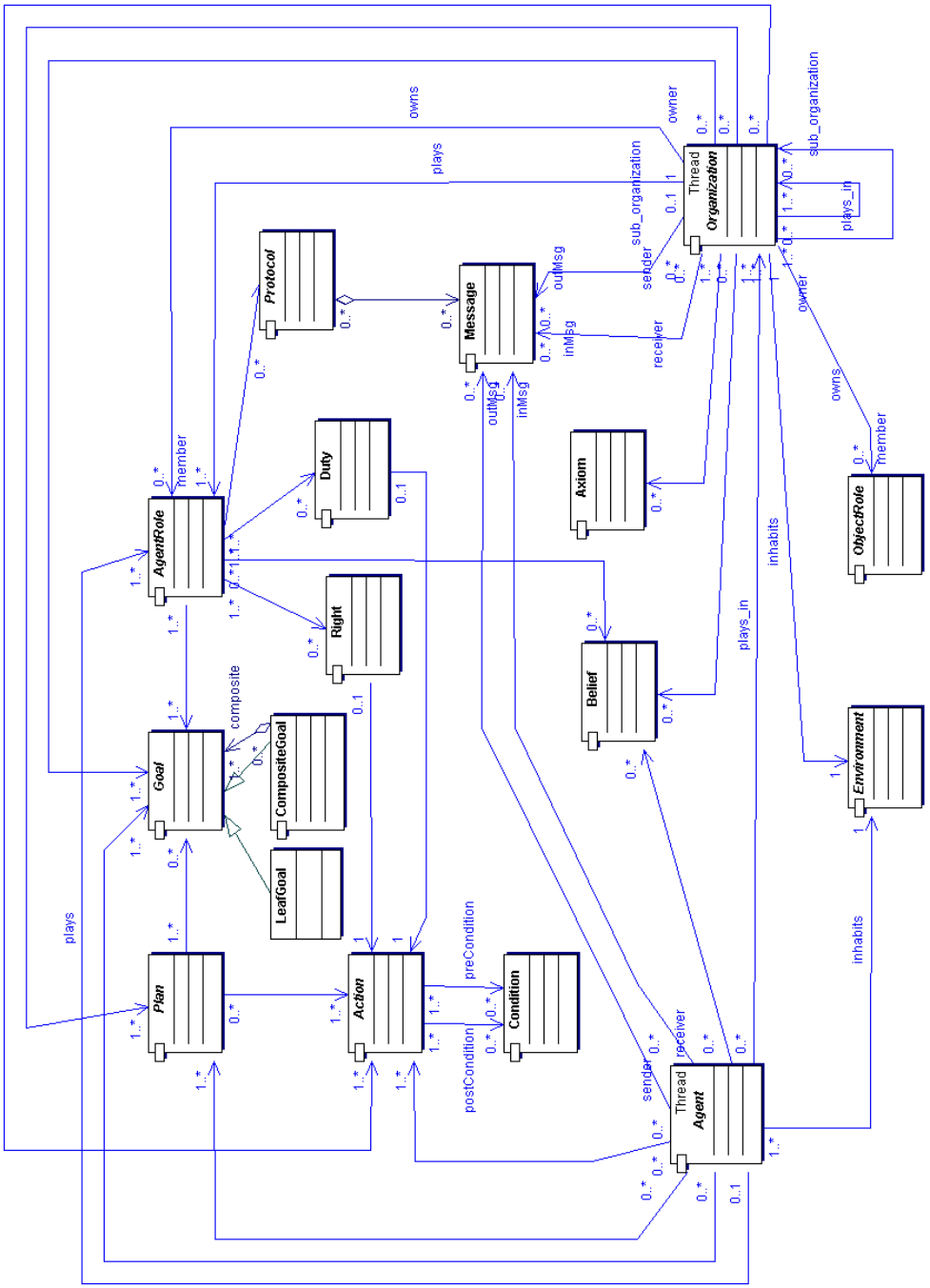

Figura 47 – A arquitetura abstrata orientada a objetos para sistemas multiagentes.

5.4.2. Regras para Entidades Dependentes do Domínio

As regras para entidades dependentes do domínio geram classes para representar as entidades descritas no domínio da aplicação. Basicamente, as classes geradas por essas regras estendem as classes abstratas definidas na Seção 5.4.1. Há um total de seis regras dependentes do domínio: regra de agente, regra

de organização, regra de papéis de agentes, regra de papéis de objetos, regra de ambientes proativos e regra de ambientes reativos. Não foi criada uma regra de objeto dependente do domínio, uma vez que as classes de objetos definidas em MAS-ML representadas em diagramas de MAS-ML são diretamente mapeadas em classes Java.

5.4.2.1. A Regra de Agentes

A regra de agentes é responsável pela criação de classes concretas a fim de representar as classes de agente definidas em uma aplicação. Para cada classe de agente definida na aplicação, uma classe OO concreta é criada com o nome do agente da aplicação. As classes que representam os agentes da aplicação estendem a classe *Agent* que herda todas as características definidas nessa classe.

Ao definir uma classe do agente usando MAS-ML, o designer descreve as ações, planos, crenças e objetivos iniciais associados a todas as instâncias do agente. Ao criar uma classe do agente OO concreta, é importante garantir que as instâncias dessa classe terão os objetivos, crenças, ações e planos definidos pelo designer. Portanto, o construtor das classes do agente OO concretas é gerado de acordo com as especificações definidas na classe do agente de MAS-ML. Ele cria instâncias de ação, plano, crença e objetivo descritas na classe do agente de MAS-ML de acordo com a estrutura das classes OO que representam essas propriedades. As instâncias de objetivo e crença são criadas instanciando as classes *LeafGoal* ou *CompositeGoal* e *Belief*, respectivamente.

A criação de instâncias de ação e plano é mais complexa do que objetivos e crenças. As classes *Plan* e *Action* são classes abstratas que devem ser especializadas de acordo com as ações e planos concretos definidos pelo agente da aplicação. As classes concretas que representam cada plano e ação são criadas com base na definição apresentada nos diagramas de MAS-ML. As instâncias de plano e ação são criadas com base nessas classes concretas. Ao instanciar uma ação, suas precondições e pós-condições são criadas com base na classe *Condition* e são associadas à ação. Ao instanciar planos, o objetivo e as ações definidos pela classe de plano no diagrama de MAS-ML são associados às instâncias de plano. Observe que os métodos `execute` que estão relacionados a planos e ações não são

implementados, uma vez que os diagramas estruturais de MAS-ML não definem a execução de um plano, nem a execução de uma ação.

Ademais, o construtor do agente deve garantir que a nova instância do agente reside em um ambiente e exerce um papel em uma organização. Portanto, ao criar uma instância do agente, é necessário definir um ambiente, um papel e uma organização. Essas informações são parâmetros de entrada do método do construtor do agente. A Figura 48 mostra parte da definição da classe *User Agent* gerada, enfatizando o método do seu construtor. O exemplo ilustrado na figura é uma saída parcial OO dependente do domínio de acordo com o exemplo de entrada de MAS-ML descrito na Seção 5.3.

```
public class User_Agent extends Agent
{
...
public User_Agent (Environment theEnvironment, Organization
initialOrg, AgentRole initialRole)
{
      ...
      this.theEnvironment = theEnvironment;
      this.theEnvironment.setAgent (this);
      setRoleBeing*Play*ed (initialRole, initialOrg);
      ...
      //creating a goal and associating with a plan
      objectGoal = new LeafGoal ("Boolean", "itemBought", "true");
      this.goals.add (objectGoal);
      objectGoal.setPlan ("Searching_seller");
      ...
      //creating a belief
      this.beliefs.add (new Belief ("Item", "item", "null"));
      ...
      //creating an action and its post-condition
      objectAction = new Search_for_seller ();
      this.actions.add (objectAction);
      objectCondition = new Condition ("Boolean", "seller",
"true");
      objectAction.setPostCondition (objectCondition);
      ...
      //creating a plan
      objectPlan = new Searching_seller ();
      this.plans.add (objectPlan);
      ...
      //associating an action with the plan
      actionAux = null;
      enumActions = this.actions.elements ();
      while (enumActions.hasMoreElements ())
      {
            actionAux = (Action) enumActions.nextElement ();
            if (actionAux.getClass ().getName ().equals
("Search_for_seller"))
            {
                  objectPlan.setAction (actionAux);
            }
      }
      //associating a goal with the plan
```

```
      goalAuxLeaf = null;
      enumGoals = this.leafGoals.elements ();
      while (enumGoals.hasMoreElements ())
      {
            goalAuxLeaf = (LeafGoal) enumGoals.nextElement ();
            if (goalAuxLeaf.getName ().equals ("itemBought"))
            {
                  objectPlan.setGoal (goalAuxLeaf);
            }
      }
}
public void run ()
{
      // TO BE IMPLEMENTED
}
}
```

Figura 48 – A classe User Agent.

5.4.2.2. A Regra de Organizações

A regra de organizações é tão complexa quanto a regra do agente. A regra de organizações é responsável pela criação de classes concretas a fim de representar as classes de organização definidas em uma aplicação. Essas classes estendem a classe abstrata *Organization* herdando todas as características definidas nela.

Ao definir uma classe de organização usando MAS-ML, o designer define os axiomas, as ações, planos, crenças e objetivos iniciais associados a todas as instâncias de organização. Ao criar uma classe de organização OO concreta, o construtor da organização deve garantir que todas as instâncias de organização terão os objetivos, crenças, ações, planos e axiomas definidos na classe de organização de MAS-ML. Portanto, o construtor das classes de organização OO concretas é gerado de acordo com as especificações definidas na classe de organização de MAS-ML. Semelhante ao que acontece com o construtor de agentes, o construtor de organizações cria instâncias de ação, plano, crença, objetivo e axioma descritos na classe de organização de MAS-ML de acordo com a estrutura das classes OO que representam essas propriedades. As instâncias de objetivo, crença e axioma são criadas instanciando as classes *LeafGoal* ou *CompositeGoal*, *Belief e Axiom*, respectivamente. A fim de criar instâncias de ação e plano, as classes abstratas *Action* e *Plan* também devem ser especializadas criando classes concretas. As instâncias de plano e ação de uma organização são criadas com base nessas classes concretas.

Um construtor de organização recebe como parâmetros de entrada o ambiente em que a organização reside, o papel inicial da organização e a organização em que exerce o papel inicial. No caso de uma organização principal, os últimos dois parâmetros são nulos porque a organização principal não exerce papéis em nenhuma organização. A Figura 49 ilustra o código gerado para uma classe de organização. A figura mostra o construtor de uma organização chamada *General Store* enfatizando a instanciação de um axioma. As demais instanciações foram omitidas porque são equivalentes àquelas em agentes.

```
public class General_Store extends Organization
{
...
public General_Store (Environment theEnvironment, AgentRole
initialRole, Organization initialOrg)
{
     ...
     //creating an axiom
     this.axioms.add (new Axiom ("boolean",
"Send_information_about_sale", "true"));
     ...
}
public void run ()
{
    // TO BE IMPLEMENTED
}
}
```

Figura 49 – A classe de organização General Store.

5.4.2.3. A Regra de Papéis de Agentes

A regra de papéis de agentes é responsável pela criação de classes concretas a fim de representar as classes de papel de agente definidas em uma aplicação. Para cada classe do papel de agente definida na aplicação, uma classe OO concreta é criada com o nome do papel de agente. Essas classes estendem a classe abstrata *Agent Role* herdando todas as características definidas nela.

Ao definir uma classe do papel de agente usando MAS-ML, o designer pode descrever protocolos, direitos, deveres, crenças e objetivos iniciais associados a todas as instâncias de papel de agente. O construtor de classes de papel de agente cria instâncias de objetivo, crença, dever, direito e protocolo descritas na classe do papel de agente de MAS-ML de acordo com a estrutura das classes OO que representam essas propriedades. As instâncias de objetivo, crença, dever e direito são criadas instanciando as classes *LeafGoal* ou *CompositeGoal*, *Belief, Duty* e *Right*, respectivamente. A fim de criar instâncias de protocolo, as classes abstratas

Protocol também devem ser especializadas criando classes concretas. As instâncias de protocolo de um papel do agente são criadas com base nessas classes concretas.

Como os papéis de agente são membros de uma organização, seu construtor recebe em um de seus parâmetros de entrada a organização da qual é uma membro. A Figura 50 ilustra a classe do papel de agente *Buyer* e seu construtor.

```
public class Buyer extends AgentRole
{
    public Buyer (Organization owner)
    {
        ...
        //creating a goal
        objectGoal = new LeafGoal ("boolean", "itemBought",
"true");
        this.goals.add (objectGoal);
        ...
        //creating a duty
        this.duties.add (new Duty ("Search_for_seller"));
        ...
        //creating a right
        this.rights.add (new Right
("Send_answer_proposal_accept"));
        ...
        //creating a protocol
        objectProtocol = new SimpleNegotiation ();
        this.protocols.add (objectProtocol);
        ...
        //creating a message
        objectMessage = new Message ("Request", "ItemDescription",
"Buyer", "Seller");
        ...
        //associating a protocol with a message
        objectProtocol.setMessage (objectMessage);
        ...
    }
}
```

Figura 50 – A classe do papel do agente Buyer.

5.4.2.4. A Regra de Papéis de Objetos

A regra de papéis de objetos é responsável pela criação de classes concretas a fim de representar as classes de papel de objeto definidas em uma aplicação. Para cada classe do papel de objeto definida na aplicação, uma classe OO concreta é criada com o nome do papel de objeto estendendo a classe abstrata *Object Role* e herdando todas as características definidas nessa classe. Além disso, essa regra define o construtor da classe do papel de objeto. Como os papéis de objeto são membros de uma organização, o construtor de papéis de objeto recebe como um de seus parâmetros de entrada a organização que possui o papel sendo criado. As

propriedades (atributos e métodos) dessa classe são criadas de acordo com sua representação nos diagramas de MAS-ML. A Figura 51 ilustra o código gerado para uma classe do papel de objeto *Desire* gerada.

```
public class Desire extends ObjectRole
{
    ...
    public void Desire (Organization owner)
    {
        setOwner (owner);
        owner.setObjectRole (this);
    }
    ...
}
```

Figura 51 – A classe do papel de objeto Desire.

5.4.2.5. A Regra de Ambientes Proativos

A regra de ambientes proativos é tão complexa quanto a regra do agente. A estrutura de um ambiente proativo é semelhante à estrutura de um agente que define objetivos, crenças, ações e planos. Portanto, é necessário instanciar os objetivos, crenças, planos e ações definidos pela entidade *proactive environment* de MAS-ML junto com a criação de classes concretas a fim de representar os ambientes proativos definidos na aplicação. As classes concretas que representam o ambiente definido na aplicação estendem a classe abstrata *Environment* e herdam todas as suas características.

A diferença entre a regra do ambiente proativo e a regra de agentes é a definição do método do construtor. Os métodos do construtor de classes concretas de ambiente não definem nenhum parâmetro de entrada. Um ambiente deve ser o primeiro elemento a ser criado em qualquer SMA e, portanto, não depende de nenhuma entidade.

5.4.2.6. A Regra de Ambientes Reativos

A regra de ambientes reativos é a regra mais simples definida pelo transformador. A estrutura de um ambiente reativo é semelhante à estrutura de uma classe que define atributos e métodos. Portanto, a regra de ambientes reativos só é responsável pela criação de classes concretas a fim de representar os ambientes reativos definidos em uma aplicação estendendo a classe abstrata *Environment*, conforme ilustrado na Figura 52.

```
import java.util.*;
```

```
public class Virtual_Marketplace extends Environment
{      ...
}
```

Figura 52 – A classe de ambiente Virtual Marketplace.

5.4.3. Regras para Relacionamentos Dependentes do Domínio

As regras para relacionamentos dependentes do domínio transformam os relacionamentos definidos nos diagramas de MAS-ML em código OO. Em Java, os relacionamentos definidos em UML (*association*, *aggregation*, *specialization* e *dependency*) já têm uma implementação padrão. Apesar de MAS-ML ter estendido esses relacionamentos a fim de modelar as interações entre abstrações de SMAs, a implementação desses relacionamentos não foi alterada.

Quase todos os relacionamentos entre duas entidades indicam o tipo de interações entre elas. Para interagir, uma entidade deve conhecer outra entidade. Portanto, quase todos os relacionamentos são representados em Java como atributos. O único relacionamento que não representa interações é o relacionamento *specialization* e, dessa forma, não é representado como um atributo. Há um conjunto de oito regras de relacionamentos.

5.4.3.1. A Regra do Relacionamento *Specialization*

Essa regra é responsável pela representação do relacionamento *specialization* no código Java gerado. Esse relacionamento identifica uma subclasse que especializa a superclasse. Ele é representado em Java pela palavra reservada `extends` usada na definição da subclasse a fim de indicar qual classe (superclasse) estende (ou especializa). Portanto, quando um relacionamento *specialization* é definido nos diagramas de MAS-ML, a classe que define a subentidade é modificada para indicar a entidade estendida usando a palavra reservada `extends` conforme ilustrado na Figura 53.

```
public class Buyer_of_Imported_Books extends Buyer
{ ...
}
```

Figura 53 – Os relacionamentos *specialization* entre Buyer of Imported Books e Buyer.

5.4.3.2. As Regras dos Relacionamentos *Association* e *Aggregation*

Essas regras são responsáveis pela representação dos relacionamentos *association* e *aggregation* no código Java gerado. O relacionamento *association* identifica as classes das entidades que podem interagir e a multiplicidade associada a elas, ou seja, o número de entidades das classes que pode interagir. Em Java, o relacionamento *association* é representado como um atributo. Um atributo é criado nas duas classes para armazenar as entidades da outra classe. As entidades de uma classe podem interagir com as entidades de outra classe, pois conhecem essas entidades. Se uma entidade de uma determinada classe pode interagir com mais de uma entidade da outra classe, deve ser usada uma estrutura para armazenar um conjunto de entidades ao definir o atributo. A Figura 54 mostra um atributo que representa o relacionamento *association* entre os papéis do agente *Buyer* e *Seller*.

```
public class Buyer extends AgentRole
{
    /**
    * @associates <{Seller}>
    * @link association
    * @label
    * @clientRole
    * @clientCardinality 0..*
    * @supplierRole
    * @supplierCardinality 0..*
    */
    protected Vector theSeller = new Vector ();
    ...
}
```

Figura 54 – O relacionamento *association* entre Buyer e Seller.

O relacionamento *aggregation* identifica a classe da entidade que representa o todo e a classe da entidade que representa as partes. Também pode ser usada a multiplicidade no relacionamento *aggregation* a fim de indicar quantas entidades podem agregar outra entidade e quantas podem ser incorporadas por uma entidade. Em Java, o relacionamento *aggregation* é representado como um atributo na classe do agregador. A Figura 55 ilustra o relacionamento *aggregation* entre as classes de papel do agente *Buyer* e *Buyer of Imported Books*.

```
import java.util.*;
public class Buyer extends AgentRole
{
    /**
    * @associates <{Buyer_of_Imported_Books}>
    * @link aggregation
```

```
    * @label
    * @clientRole
    * @clientCardinality 1
    * @supplierRole
    * @supplierCardinality 0..*
    */
    protected Vector theBuyer_of_Imported_Books = new Vector ();
    ...
}
```

Figura 55 – O relacionamento *aggregation* entre Buyer e Buyer of Imported Books.

5.4.3.3. A Regra do Relacionamento *Control*

Essa regra é responsável pela representação do relacionamento *control* no código Java gerado. Esse relacionamento identifica a classe das entidades que representa os controladores e a classe das entidades que são entidades controladas. Também pode ser usada a multiplicidade no relacionamento *control* a fim de representar quantas entidades podem ser controladas por uma entidade e quantas podem controlar uma entidade. Ao gerar código a partir desse relacionamento descrito em diagramas de MAS-ML, um atributo é criado na classe das entidades que são controladoras e um atributo, na classe das entidades que são as controladas. Os controladores devem conhecer quem estão controlando, e os controlados devem conhecer quem são seus controladores a fim de responder a eles. A Figura 56 e Figura 57 ilustram os atributos criados para representar um relacionamento control.

```
public class Market_of_Special_Goods extends AgentRole
{
    /**
    * @associates <{Seller_of_Imported_Books}>
    * @link association
    * @label
    * @clientRole controller
    * @clientCardinality 1
    * @supplierRole controlled
    * @supplierCardinality 0..*
    */
    protected Vector theSeller_of_Imported_Books = new Vector ();
    ...
}
```

Figura 56 – O relacionamento *control* ilustrado em Market of Special Goods que controla Seller of Imported Books.

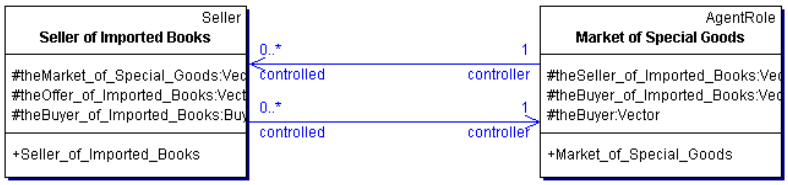

Figura 57 – O relacionamento *control*.

5.4.3.4. A Regra do Relacionamento *Dependency*

Essa regra é responsável pela representação do relacionamento *dependency* no código Java gerado. Esse relacionamento identifica a classe das entidades que são os clientes e a classe das entidades que são os fornecedores. Em Java, o relacionamento *dependency* normalmente não é representado por atributos globais, ou seja, atributos de uma classe. Os fornecedores de um cliente estão implícitos no código. O relacionamento *dependency* entre um cliente e um fornecedor pode ser identificado normalmente em duas situações: os fornecedores estão indicados como atributos locais ou como parâmetros de entrada de um método. Como não é possível encontrar onde os fornecedores são acessados ao usar as informações disponíveis nos diagramas estruturais, também não é possível modificar o código gerado criando atributos locais ou parâmetros de entrada de métodos.

Apesar de não ser possível encontrar no código onde os fornecedores são acessados, não acreditamos ser importante representar graficamente esse relacionamento. A fim de exemplificar os relacionamentos *dependency*, as classes das entidades que são clientes são modificadas pela inclusão de atributos comentados. Usando esses atributos, é possível representar graficamente o relacionamento sem alterar as características da classe. Como o atributo é comentado, o número de atributos da classe não foi modificado.

Os atributos comentados foram criados com base na nomenclatura usada pela ferramenta gráfica Together (Together, 2004) [5]. Together usa esses atributos para representar graficamente todos os relacionamentos *dependency*. A Figura 58 exemplifica o uso de atributos comentados a fim de implementar esse relacionamento.

[5] Together foi usado porque gera modelos de UML a partir do código Java.

```
public class Client
{
    /**
    * @associates <{Supplier}>
    * @link dependency
    * @supplierCardinality 0..*
    */
    /*# Vector theSupplier; */
    ...
}
```

Figura 58 – O relacionamento *dependency* entre Client e Supplier.

5.4.3.5. A Regra do Relacionamento *Inhabit*

Essa regra é responsável pela representação do relacionamento *inhabit* no código Java gerado. O relacionamento *inhabit* identifica a classe das entidades que são os habitats e a classe das entidades que são os cidadãos.

Todos os agentes, as organizações e os objetos residem em um ambiente. A fim de implementar esse relacionamento independente do domínio entre agentes e um ambiente, ao gerar a classe abstrata *Agent* usando a regra de agentes básica, independente do domínio, um atributo para representar o ambiente foi definido na classe abstrata *Agent*, associando essa classe à classe abstrata *Environment*. Assim, as instâncias das classes concretas de agente (classes da aplicação) que estendem *Agent* são automaticamente relacionadas às instâncias das classes concretas de ambiente que estendem *Environment*. O mesmo tipo de atributo foi usado para relacionar organizações e ambientes ao gerar a classe abstrata *Organization* por meio da regra de organizações básica, independente do domínio. Entretanto, o atributo não pode ser definido para objetos porque a superclasse das classes é a classe *Object* definida pela linguagem Java que não pretendemos modificar.

Ademais, os atributos definidos na classe abstrata *Environment* também foram criados para indicar a associação entre a classe abstrata e todos os agentes (representados por uma associação com a classe abstrata *Agent*), as organizações (representadas por uma associação com a classe abstrata *Organization*) e objetos (representados por uma associação com a classe Java *Object*).

Como as classes abstratas *Agent*, *Organization* e *Environment* estão ligadas pelo relacionamento *inhabit*, não há a necessidade de representar o relacionamento *inhabit* entre as classes concretas definidas em aplicações. Apesar de não haver essa necessidade, foram definidos atributos comentados associados a

classes do agente concretas, classe de organizações, classes e classes de ambiente, ao gerar código a partir dos relacionamentos *inhabit* descritos em aplicações. A Figura 59 exemplifica o uso dos atributos comentados a fim de implementar o relacionamento *inhabit*, e a Figura 60 ilustra graficamente duas classes do agente concretas ligadas a uma classe de ambiente concreta pelo relacionamento *inhabit*.

```
public class User_Agent extends Agent
{
    /**
    * @link dependency
    * @label inhabit
    */
    /*# Virtual_Market_Place theEnvironment; */
    ...
}
```

Figura 59 – O relacionamento *inhabit* entre User Agent e Virtual marketplace.

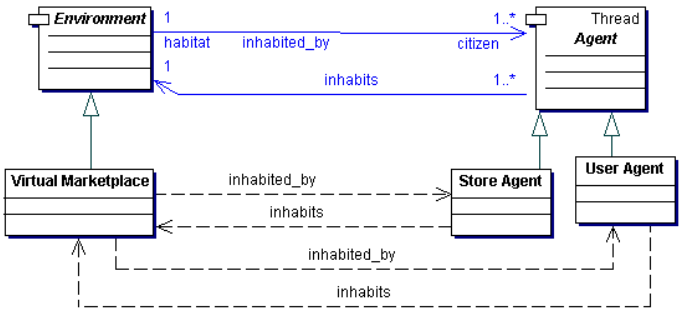

Figura 60 – O relacionamento *inhabit*.

5.4.3.6. A Regra do Relacionamento *Play*

Essa regra é responsável pela representação do relacionamento *play* no código Java gerado. O relacionamento *play* identifica a classe de entidades que exercem papéis, a classe dos papéis que estão sendo exercidos e a classe da organização em que os papéis estão sendo exercidos.

Todos os agentes e as suborganizações exercem pelo menos um papel em uma organização. A fim de implementar esse relacionamento independente do domínio entre agentes e papéis, dois atributos foram criados ao gerar a classe abstrata *Agent* usando a regra de agentes básica, independente do domínio. Um atributo representa os papéis associando a classe abstrata *Agent* à classe abstrata *Agent Role*, e o outro atributo representa as organizações em que os papéis são exercidos associando a classe abstrata *Agent* à classe abstrata *Organization*. Dessa forma, as instâncias de agente de classes do agente concretas que estendem *Agent* são automaticamente relacionadas a instâncias de papel de classes de papel

concretas que estendem a classe *Agent Role* e a instância de organização das classes de organização concretas que estendem a classe *Organization*.

O mesmo tipo de atributos foi usado para relacionar suborganizações, papéis e as organizações em que os papéis estão sendo exercidos ao gerar a classe abstrata *Organization* usando a regra de organizações básica, independente do domínio. Ademais, os atributos definidos na classe abstrata *Agent Role* também foram criados pela regra para papel de agente básica independente do domínio para indicar a associação entre os papéis e todos os agentes (representados por uma associação com a classe abstrata *Agent*) e todas as organizações (representadas por uma associação com a classe abstrata *Organization*).

Apesar de não haver a necessidade de representar os relacionamentos *play* concretos (porque são abstratamente representados pelos atributos usados nas classes abstratas), um atributo comentado associado a classes do agente concretas foi definido ao gerar código a partir de um relacionamento *play* descrito em MAS-ML. Esse atributo representa o relacionamento concreto entre a classe do agente concreta e a classe do papel do agente concreta. A Figura 61 exemplifica o uso dos atributos comentados para implementar o relacionamento *play* entre a classe do agente concreta *Store_Agent* e a classe do papel do agente concreta *Seller*. A Figura 62 ilustra graficamente o relacionamento *play*.

```
public class Store_Agent extends Agent
{
    /**
    * @associates <{Seller}>
    * @link dependency
    * @label play
    */
    /*# Vector rolesBeingPlayed; */
    ...
}
```

Figura 61 – O relacionamento *play* entre Store Agent e o papel do agente Seller.

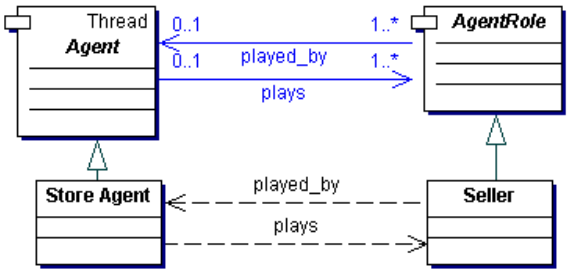

Figura 62 – O relacionamento *play* relacionando um agente a um papel do agente.

Objetos também exercem papéis em organizações. Contudo, os objetos não conhecem os papéis que estão exercendo. Portanto, as classes não têm qualquer atributo que indique os papéis que suas instâncias estão exercendo. Por outro lado, cada papel conhece o objeto que está exercendo o papel. Ao gerar código a partir de um relacionamento *play* usado entre uma classe de objeto e uma classe do papel de objeto, um atributo e um método são criados na classe do papel de objeto. O método define o objeto que está exercendo o papel usando o atributo. O atributo está ilustrado em Figura 63.

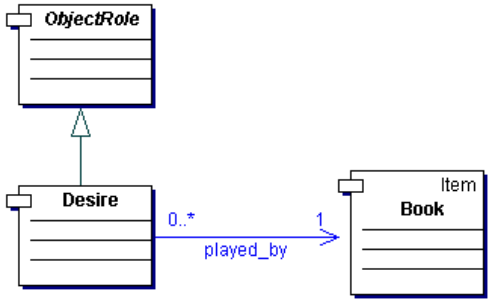

Figura 63 – O relacionamento *play* relacionando uma classe a um papel de objeto.

5.4.3.7. A Regra do Relacionamento *Ownership*

Essa regra é responsável pela representação do relacionamento *ownership* no código Java gerado. Esse relacionamento identifica a classe das organizações que são proprietárias e a classe dos papéis (papéis de objeto e agente) que são os membros.

Todos os agentes e os papéis de objeto são membros de uma organização. A fim de implementar esse relacionamento independente do domínio entre organizações e papéis, dois atributos foram criados ao gerar a classe abstrata *Organization* usando a regra de organização básica, independente do domínio. Um atributo representa os papéis do agente associando a classe abstrata *Organization* à classe abstrata *Agent Role*, e o outro atributo representa os papéis de objeto associando a classe abstrata *Organization* à classe abstrata *Object Role*. Assim, as instâncias de organização das classes de organização concretas que estendem *Organization* estão automaticamente relacionadas às instâncias de papéis das classes de papel concretas que estendem as classes *Agent Role* ou *Object Role*.

Apesar de não haver a necessidade de representar os relacionamentos *ownership* concretos (porque são abstratamente representados pelos atributos usados nas classes abstratas), atributos comentados associados às classes de organização concretas foram definidos ao gerar código a partir de um relacionamento *ownership* descrito em MAS-ML, conforme ilustrado na Figura 64.

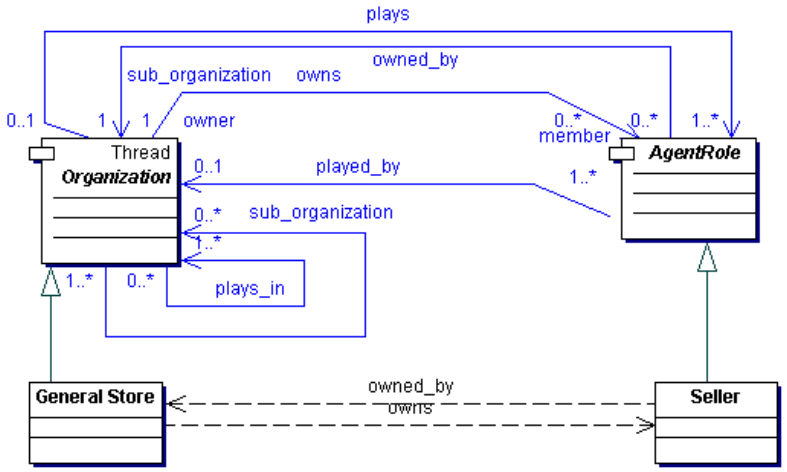

Figura 64 – O relacionamento *ownership*.

5.5. Discussão

Nossa abordagem para o design de uma arquitetura abstrata OO para sistemas multiagentes não considera a extensão da arquitetura abstrata da FIPA disponível (FIPA, 2000). A arquitetura da FIPA não foi estendida uma vez que a FIPA e a nossa abordagem têm objetivos diferentes. Nosso objetivo ao desenvolver a arquitetura abstrata era criar uma arquitetura abstrata OO que contemplasse todos os conceitos definidos no metamodelo de MAS-ML. Concentramo-nos na descrição de uma arquitetura abstrata OO que representava as entidades de MAS-ML, suas propriedades e relacionamentos pelo uso de classes OO. Não pretendíamos criar uma arquitetura que, por exemplo, especificasse as interações entre as entidades. Portanto, não geramos código a partir dos diagramas de seqüência propostos.

Ao contrário, o foco primário da arquitetura abstrata da FIPA é criar uma troca de mensagens semanticamente significativa entre agentes (interações) o que pode ser alcançado pelo uso de diferentes meios de transporte de mensagens,

diferentes linguagens de comunicação de agentes ou diferentes linguagens de conteúdo (FIPA Architecture, 2004). A arquitetura da FIPA não inclui abstrações como organização e papel e, assim, as interações entre essas abstrações e agentes não são definidas. O escopo da arquitetura da FIPA inclui: um modelo de serviços e descoberta de serviços disponíveis para agentes e outros serviços, interoperabilidade no transporte de mensagens, oferecendo suporte a diversas formas de representações ACL, à linguagem de conteúdo e a várias representações de serviços de diretório.

Ao desenvolver o transformador MAS-ML2Java, não pretendemos criar uma ferramenta que gerasse automaticamente código a partir de todos os modelos de MAS-ML. A fim de criar essa ferramenta, seria necessário gerar código a partir de diagramas de seqüência. Para gerar código dessa forma, seria importante criar uma arquitetura que especificasse as interações entre abstrações de SMAs. Como a arquitetura da FIPA refere-se à interação entre agentes, poderia ser usada para gerar código a partir do diagrama de seqüência. Poderiam ser usadas duas abordagens diferentes. Nossa arquitetura abstrata poderia ser estendida para incorporar as características descritas na arquitetura da FIPA, ou a arquitetura da FIPA poderia ser estendida a fim de modelar as abstrações de MAS-ML.

Ao gerar código a partir de modelos de MAS-ML, nosso propósito foi produzir uma implementação independente da arquitetura. Dessa forma, arquiteturas concretas como (Bradshaw, 1996; Aglets, 2004; Jade, 2004) não foram usadas na transformação proposta. O código de implementação gerado deveria refletir o design de SMAs de uma aplicação. Pretendemos gerar um design OO em que o designer de MAS-ML pudesse facilmente identificar as entidades, suas propriedades e relacionamentos, conforme ilustrado nos modelos de MAS-ML. Esse design OO deve ser tão simples quanto possível, ou seja, não ter mais classes do que o conjunto mínimo de classes usado para representar as abstrações. Se arquiteturas concretas fossem usadas, seria necessário criar muitos modelos de componente que não seriam diretamente relacionados à definição das abstrações. Arquiteturas concretas definem, por exemplo, modelos de componente para implementar os princípios de agentes, o registro de agentes em um ambiente e os protocolos de comunicação e as linguagens de comunicação usadas para enviar e receber mensagens.

A fim de usar uma arquitetura concreta ao gerar código a partir de modelos de MAS-ML, seria necessário estender a arquitetura concreta incorporando as abstrações definidas em MAS-ML. A extensão da arquitetura concreta tem de ser desenvolvida com base na arquitetura abstrata de MAS-ML. A arquitetura concreta estendida seria uma especialização da arquitetura abstrata de MAS-ML.

Com o objetivo de descrever textualmente as informações gráficas apresentadas em diagramas estruturais de MAS-ML, uma gramática de MAS-ML foi desenvolvida. Usando essa gramática, é possível descrever todas as informações apresentadas em um diagrama estrutural de MAS-ML, porque descreve todas as entidades de MAS-ML, suas propriedades e todos os relacionamentos. Em vez de criar a gramática de MAS-ML, a linguagem XMI (XMI, 2004) poderia ter sido usada. XMI é uma linguagem de marcação estendida (extended markup language) para o intercâmbio de documentos, usada por UML para descrever os diagramas de UML. Usando XMI, é possível descrever as informações disponíveis em qualquer diagrama de UML. Para usar XMI, seria necessário estender UML DTD, que representa o metamodelo de UML. UML DTD deve ser estendido para também descrever as informações disponíveis no metamodelo de MAS-ML gerando MAS-ML DTD. Depois de estender UML DTD, a gramática deve ser descrita usando a linguagem TXL. Ademais, as regras descritas com base na gramática de MAS-ML devem ser modificadas para transformar instâncias de XMI descritas usando MAS-ML DTD em Java.

Como nosso objetivo ao realizar essa transformação era ilustrar as entidades de MAS-ML, propriedades e relacionamentos em uma linguagem OO, o uso de XMI nos ofereceria apenas vantagens limitadas. O benefício de XMI é a especificação de todas as abstrações modeladas no diagrama de UML. Ferramentas como Together geram diagramas OO e código a partir de uma aplicação descrita usando XMI. Não pretendemos repetir o trabalho que Together já realiza. Decidimos nos concentrar na geração de código a partir de abstrações de MAS-ML. Se UML DTD tivesse sido estendida, poderíamos gerar código para todas as abstrações de MAS-ML e para todas as abstrações de UML. Seria interessante estender UML DTD se o objetivo fosse criar uma ferramenta de modelagem de MAS-ML que incluísse um ambiente para modelar diagramas de MAS-ML e uma máquina geradora desde diagramas de MAS-ML até o código, como Together para UML (consulte a Seção 7.2).