Establishing Operational Models for Dynamic Compilation in a Simulation Platform

Nghi Quang Huynh[1(✉)], Tram Huynh Vo[2], Hiep Xuan Huynh[1], and Alexis Drogoul[3]

[1] DREAM-CTU/IRD, CICT-CTU, Cantho, Vietnam
{hqnghi,hxhiep}@ctu.edu.vn
[2] Software Engineering Department, CICT-CTU, Cantho, Vietnam
vhtram@ctu.edu.vn
[3] UMI 209 UMMISCO, IRD, Hanoi, Vietnam
alexis.drogoul@ird.fr

Abstract. In this paper we introduce a new approach to dynamic converting conceptual models in a simulation platform as the GAMA platform (represented in form of GAML syntax) into corresponding operational models (represented in form of Java syntax). This approach aims at providing a more flexible solutions to actual simulation models implemented in a simulation platform as the GAMA. This new approach will facilitate the exhibits of a simulation platform to work with different types of simulation models represented in different forms of syntax.

Keywords: Domain-specific language · Operational model · Simulation platform · Dynamic compilation

1 Introduction

GAML [26] is the modeling language of the GAMA platform [22], which was based on XML syntax [3]. This language designed as a simple scripting language for platform of simulation, has grown into a general language dedicated to the modeling (modeling language [24]). From its appearing, the diversity of multi modeling language resulted a difficult challenge to reuse models between platforms, to increase numbers of new features and to have a large developer community. In this context, we propose a generic method of establishing operational model in simulation platforms. Since version 1.4, GAMA has had a developed environment built with the technology XText [4](based on ANTLR [17] grammar) and is considered, ultimately, able to build models directly in Java [5].

The main objective of our work is to convert from a GAML model, represent the Abstract Syntax Tree [11] as one Java class or even one Java project Java which can run alone without the GAMA compiler. From this Java model, we can have another way to combine the models, to attach plugins, and of course to have inheritance of Java. In addition, running the Java models will take less time and memories, gain speed of models processing. With this work, we can

P.C. Vinh et al. (Eds.): ICTCC 2014, LNICST 144, pp. 117–131, 2015.
DOI: 10.1007/978-3-319-15392-6_12

take a review of what the current grammar can do now, by translation to Java syntax, review the advantages and inconveniences between Java platform and GAMA platform.

This paper will be represented in six parts. The second part introduces related works of three type of model in modeling. In the third part, we talk about our methodology for establishing operational model, due to algorithm of traversing Abstract Syntax Tree and our strategies of creating Java syntax. In the fourth part, this method was implemented into a simulation platform GAMA. Then we'll tell you how our experimental results have been taken in the fifth part, based on converting Bug model with description of model, species, attributes and behaviors. Finally, in sixth part, we give information about our results and ongoing work.

2 Related Work

2.1 From XML to Operational Model

XML [3] has usually been used in modeling domains, which can be considered as an flexible integration approach into modeling and simulation systems [19]. As the effort towards standardization of formalism representations, based on the XML schema definition of a formalism, a binding compiler generates model classes that support the user in constructing models according to the formalism. Although simulators could be built for these declarative model descriptions, they would be hardly efficient. To this end, a separate transformation component is required according to manually pre-defined XML schemas [7].

Other effort of using XML in modeling and simulation is to compose simulations from XML-specified model components [20]. It presents the realization of a component framework that can be added as an additional layer on the top of simulation systems. It builds upon platform independent specifications of components in XML to evaluate dependent relationships and parameters during composition. The process of composition is split up into four stages. Starting from XML documents component instances are created. These can be customized and arranged to form a composition. Finally, the composition is transformed to an executable simulation model.

in the DEVS community , a current notable effort is to provide a worldwide platform for distributed modeling and simulation based on web services and related technologies[24].This infrastructure will allow the sharing and reuse of simulation models and experiments and will permit attacking different aspects of interoperability at the right level of abstraction: the simulation-based interoperability at the level of the data transfer among components in a distributed simulation, the model-based interoperability to share models and experiments. An essential requirement is that a common, unique and complete representation must be adopted to store, retrieve, share and make interoperable simulation models. The author represent all the aspects included in all possible use cases and proposes an XML-based language that can serve as a basis for defining a standard for distributed simulation linking DEVS and non-DEVS simulations.

2.2 From Domain-Specific Language to Operational Model

Domain-specific languages (DSL) [16] are languages tailored to a specific application domain. They offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages in their domain of application. DSL development is hard, and it requires both domain knowledge and language development expertise. Few people have all these two. Not surprisingly, the decision to develop a DSL is often postponed indefinitely, if considered at all, and most DSLs never get beyond the application library stage.

Aspen (Abstract Scalable Performance Engineering Notation) [21] fills an important gap in existing performance modeling techniques and is designed to enable rapidly exploration of new algorithms and architectures. It includes a formal specification of an application's performance behavior and an abstract machine's model. It provides an overview of Aspen's features and demonstrate how it can be used to express a performance model for a three dimensional Fast Fourier Transform [8] . It demonstrates the composability and modularity of Aspen by importing and reusing the FFT model in a molecular dynamics model. It have also created a number of tools that allow scientists to balance application and system factors quickly and accurately.

2.3 Dynamic Compilation

Dynamic compilation [23] is a process used by some programming language implementations to gain performance during program execution. Although the technique is originated in the Self [6] programming language, the best-known language that uses this technique is Java. Since the machine code emitted by a dynamic compiler is constructed and optimized at program runtime, the use of dynamic compilation enables optimizations for efficiency not available to compiled programs except through code duplication or meta-programming. Runtime environments using dynamic compilation typically have programs run slowly for the first few minutes, and after that, most of the compilation and recompilation is done and it runs quickly. Due to this initial performance lag, dynamic compilation is undesirable in certain cases. In most implementations of dynamic compilation, some optimizations that could be done at the initial compile time are delayed until further compilation at run time, causing further unnecessary slowdowns. Just in time compilation is a form of dynamic compilation.

Dynamic compilation bring more and more fast, effective and optimization values [1] [12] due to invariant data computed at run-time. Using the values of these run-time constants, a dynamic compiler can eliminate their memory loads, perform constant propagation and folding, remove branches they determine, and fully unroll loops they bound.

Dynamic compilation increases Java virtual machine (JVM) performance [15] because running compiled codes is faster than interpreting Java byte-codes. However, inappropriate decision on dynamic compilation may degrade performance owing to compilation overhead. A good heuristic algorithm for dynamic compilation should achieve an appropriate balance between compilation overhead

and performance gain in each method invocation sequence. A method-size and execution-time heuristic algorithm is proposed in the study.

In brief, dynamic compilation have fully of benefits investing [14]. Dynamic compilation is typically performed in a separate thread, asynchronously with the remaining application threads. It explores a number of issues surrounding asynchronous dynamic compilation in a virtual machine by describing the shortcomings of current approaches and demonstrate their potential to perform poorly under certain conditions. It shown the importance of ensuring a level of utilization for the compilation thread and empirically validate this in a production virtual machine on a large benchmark suite; beside evaluation a range of compilation thread utilizations and quantify the effect on both performance and pause times.

3 From Domain-Specific Language to Operational Model

3.1 What Is a Model?

Model [18], especially scientific model [25], is an abstract construction, that allows to comprehensive functions of the reference system by answering one scientific question. It is an simplify representation of the reference system, relying on generic theory and can be expressed in one specific language which called Modeling language [9].

3.2 Model in a Simulation Platform

Model in simulation platform, especially GAMA, contains 4 main sections:

Global section contains all declaration of variables, parameters at global scope. These declarations can be used anywhere in model. This section also contains starting point when a model is executed, *init* block. This "global" section defines the "world" agent, a special agent of a GAMA model. We can define variables and behaviors for the "world" agent. Variables of "world" agent are global variables thus can be referred by agents of other species or other places in the model source code.

Environment section contains informations about topology which are used by agents. GAMA supports three types of topologies for environments: continuous, grid and graph. By default, the world agent (i.e. the global agent that contains all of the other agents) has a continuous topology. This section could include the definition of one or several environments with grid topology.

Entities section defines of all species which are placed into this section. A model can contain any number of species. Species are used to specify the structure and behaviors of agents. Although the definitions below apply to all the species, some of them require specific declarations: the species of the world and the species of the environmental places.

Experiment section defines experiments to run. Two kinds of experiment are supported: gui (graphic user interface, which displays its input parameters and outputs) and batch (Allows to setup a series of simulations without graphical interface).

3.3 Abstract Syntax Tree

An abstract syntax tree (AST) [11] , is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is 'abstract' and doesn't represent every detail appearing in the real syntax. For instance, grouping parentheses are implicit in the tree structure, and a syntactic construction like an if-condition-then expression may be denoted by means of a single node with two branches. This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees, which are often built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis. Abstract syntax trees are also used in program analysis and program transformation systems.

3.4 AST of Operational Model

The current version of GAMA contains five main types of node in AST, with respect to meta-model of Multi-agent systems. There are:

– Model Description: Root node of a model. It contains others children nodes. This "global" section defines the "world" agent, a special agent of a GAMA model. We can define variables and behaviors for the "world" agent. Variables of "world" agent are global variables thus can referred by agents of other species or other places in the model source code.

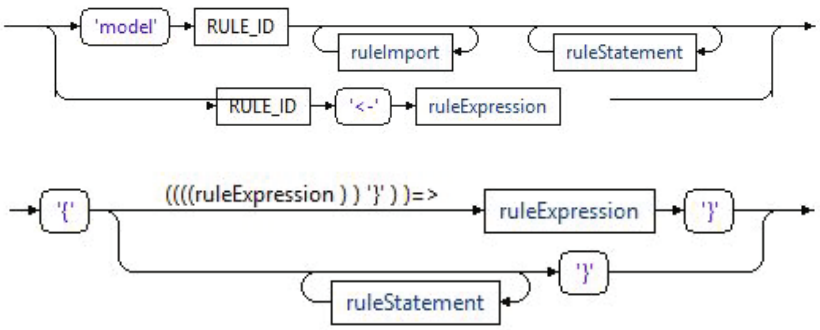

Fig. 1. AST of GAML syntax

– Experiment Description: this type of node describes the inputs, outputs parameters, the way that model would be simulated. Based on concept of MAS, Experiment can be a special species in model.

- Species Description: due to multi-scale hierarchy, a species could be a child of another species, models, or even experiments
- Type Description: more likely in a programming language, we have several base data types, integer, string, double. In fact, modelers can define themself theirs own data types, which is related to complex structures. Especially a species could be a data type, to be declared and assigned as variable later.
- Statement Description: includes simple statement and complex (sequence) statement. It's correspond with the smallest element of programming language. A species can thus contain several statements representing different behaviors that agents of the species can execute. GAMA offers a statement framework that facilitates the extension, i.e., the introduction of new types of agent's behavior. Developer can extend, in case of new needed appear, a new Statement by implement this class.
- Variable Description: In this declaration, information of variables data type refers to the name of a built-in type or a species declared in the model. The value of name can be any combination of letters and digits that does not begin with a digit and that follows certain rules. If the value of the variable is not intended to change over time, it is preferable to declare it as a constant in order to avoid any surprise (and to optimize, a little bit, the compiler's work). This is done with the **const** attribute set to true (if const is not declared, it is considered as false by default):

An example of AST for operational model in GAMA can be found in fig.2. On the right, It's an example in simple model of GAMA. In **entities** section, it's declared 2 species which are type of SpeciesDescription node in AST. In species definition, user can define other children nodes as VariableDescription, TypeDescription, StatementDescription. This example shows the usage of built-in TypeDescription of Integer number as **int** keyword. And the most important description, species have several Statement to do behaviors, e.g. **move** behavior in Ant species, and **hunt** behaviors in Predator.

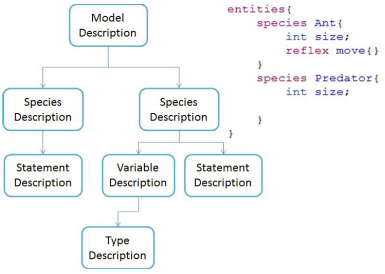

Fig. 2. AST of operational model in GAMA

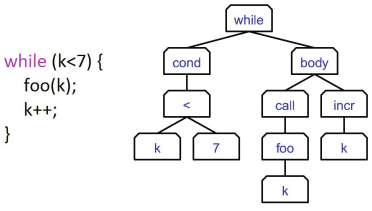

Fig. 3. AST of Java statement

3.5 Converting from GAML Syntax to Operational Model (with Java Syntax)

As both GAMA model and Java program base on a tree (Abstract Syntax Tree), we use the algorithm of traversing to explore AST.

Recursivement, we traverse the AST, from root node. At each node, by considering node's type, compiler will translate into correspond Java syntax. Following pseudo-code describe whole progress.

This function builds a abstract syntax tree which contains each node as Java syntax. It takes input as a GAML_Node, which can be following type: ModelDescription, SpeciesDescription, TypeDescription, VariableDescription, Symbol Description. At first, an empty tree would be created in local scope, with root node that will be created in next steps. This tree has attached the node which was translated in Java syntax. Next, the type of the current node was returned into variable *t*. Regarding value of this variable, compiler gets pre-defined template as string *tplt*. This template is input parameter of merging method, described in Fig. 5. This method manipulates information of the current node and template *tplt* by combining together at needed XML tag. The output is considered as root node of local tree. The loop of each child node do a recursive calling itself. After calling, root node attachs these results to child node. Finally, method returns thes whole structure tree in Java syntax.

```
1  Function Build Java Tree ( GAML_Node gNode) {
2      Let JavaTree = new tree
3      Let t= type of gNode
4      String tplt=Generate java syntax correspond with type t
5      Node root = Merging description of gNode with template tplt
6      For each child C in gNode
7      {
8          Let tree T = Build JavaTree (C)
9          Add T as child of root
10     }
11     Add tmp as root node of JavaTree
12     Return JavaTree
13 }
```

Fig. 4. Algorithm of traversing on Abstract Syntax Tree, applying to convert from GAML to Java

3.6 Generate Java Syntax

There are 2 types of generating Java syntax. The first way is to use static template, which is pre-defined by programmer. It is used to apply on non-volatile type, like Model, Species's Description, corresponding with a package, a class in Java. The second way is to use more informations inside each node of AST. It's Java annotation, declared for operator, statement and other components.

```
Function Build Java Tree ( GAML_Node gNode) {
    Let JavaTree = new tree
    Let t= type of gNode
    String tplt=Generate java syntax correspond with type t
    Node root = Merging description of gNode with template tplt
    For each child C in gNode
    {
        Let tree T = Build JavaTree (C)
        Add T as child of root
    }
    Add tmp as root node of JavaTree
    Return JavaTree
}
```

Fig. 5. Merging GAML with templates to get Java syntax

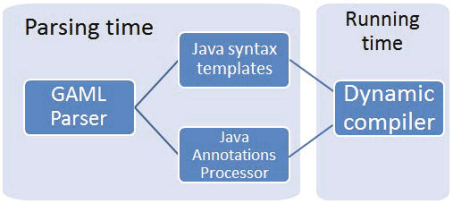

Fig. 6. Modules of G2J plugin

4 G2J Plugin

To facilitate the whole process of establishing, and make it re-usable, we implemented our method as plugin in GAMA platform. This plugin named G2J (fig.6) (GAML to Java) which is implemented in Java language. This plugin contains 4 following modules: GAML Parser, Java syntax template, Java Annotation Processor, Dynamic Compiler, which are separated into 2 phases : parsing and running. In parsing phase, it takes into account 2 process. Firstly, GAML parser is integrated into parsing process of ANTLR as grammar syntax processing. The output of GAML parser is transfered to Java Syntax Templates, and at the same time to Java Annotations Processor. These two module work in collaboration with each other to establish a complete Java Syntax Tree with the most comfortable solutions. After the first phase, Java Syntax Tree uses as input of Dynamic Compiler, which pre-compile to byte code (.class) of Java.

4.1 GAML Parser

This module implements algorithm to traversing AST in fig.4 and builds the Java syntax tree. The input of this module is a Description tree, which is described in part 3. This tree contains several types of nodes. This module is injected into parsing process of GAMA. It provides method parseTree() to call algorithm of traversing tree, return Java syntax tree.

4.2 Java Syntax Template

This module aims at reproduce corresponding Java syntax, due to pre-defined XML templates. It uses an likely-XML parser. With informations of a node from Description Tree, it get XML node in templates.

4.3 Java Annotations Processor

Beside the solution of using pre-defined templates, this module is considered as pre-processer of all Java annotation in all modules, plugins, to build additional information. These information will be used in GAML parser when traversing AST, or be combined with XML templates when calling module 2. This module provides more flexibility, and reusable solutions, in case of pre-defined templates couldn't suitable with large amount of evolving syntax in GAMA.

4.4 Java Dynamic Compiler

In addition to GAMA compiler, we have used a Java Dynamic compiler [2] [10] [13], to compile from Java syntax tree to Java byte-code class. This class is taken into account when we launch the simulation, instead of re-compiling original Abstract Syntax tree each time. In this part we introduce the method to compile the Java model and attach it to the execution tree. When we finished this paper, we had succeeded to translate only the Species entity to Java, by adding a boolean keyword named **compiled**, to tell the compiler if it must use the Java class instead of original species class in the model description tree. After having created the Java class, we use a Java plugin,batch compiler (org.eclipse.jdt.core), to compile the Java class to byte code, and load it to memory, on run time.

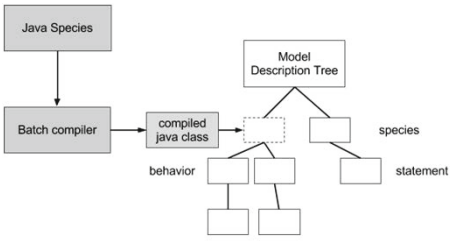

Fig. 7. Compilation process in Java species instead of GAML species

5 Experiments

By applying our manner of establishing operational, we consider below our experimental result in a simple model in simulation platform GAMA. We do some experiences on a simple model which contains **model** and **entities** section. It shows the establishment from GAML to Java on declaration of a **model**, a **species**, which demonstrates the usage of Java Syntax Templates, an **operator** and a **statement**, which are supposed to use Java Annotations Processor.

5.1 Data Described

Let's consider the model below in GAMA platform, it describes a bug which can change its color in a square environment. This species has one attribute, mycolor, representing its body's color. This bug shows itself as a circle of which the radius is 10 units and the color is green as default. At each simulation step, the attribute mycolor will change to a random rgb value, thanks to operator rnd(255). This action is a reaction of species basing on reflex architecture (line 11), which will happen if it reaches the condition following keyword **when**. In this case, it always happens because the condition is always return true.

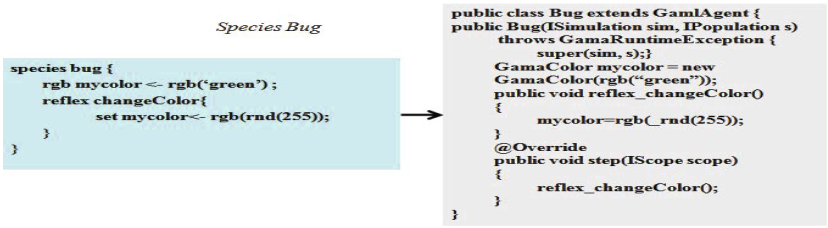

Fig. 8. Definition of Species Bug in GAML and Java

5.2 Convert Domain-Specific Language to Operational Model

By applying algorithm as Fig. 4, there are 3 following cases that we consider as main points.

Convert Species and Model. At first, by regarding the proper syntax of GAML and Java defining Species and Model, we create manually an XML template, content format of these two descriptions. Then when compiler traverses each nodes of constructed Abstract Syntax Tree, we detect the type of the current node. If it's Model node, we look into template to get corresponding XML node. In this case, a Java class would be returned which have class name be name of Model node, extended class ModelDescription to have all default predefined parameters, skills, behaviors. In Fig. 10, we have a normal declaration of model. It contains keyword model, followed by the model's name as a string. This string will be used in place of tag name in its corresponding template. After combination, compiler create an empty Java class. Body of this class will be completed later. When compiler encounters a node with type Species, process flow is much similar with previous type (Model), except some varied in extending class SpeciesDescription. This predefined class contents all must-have attributes of formal Species, ex. location, size, shape... (Fig. 11).

Applying the same manner to some more type of node in AST, we get an advantage of possibility to translate all GAML syntax to Java syntax. But it's a big challenge for modelers to create and develop new operators, skills, behaviors.

Fig. 9. Converting cases from GAML to Java

Beside of theirs implementations in Java, they must create them-self template for their new things. The job to read and create all correspond templates must take too much time, and actually in future, it will not adapt well and quickly (depend on human-additions templates) when GAML syntax has evolved.

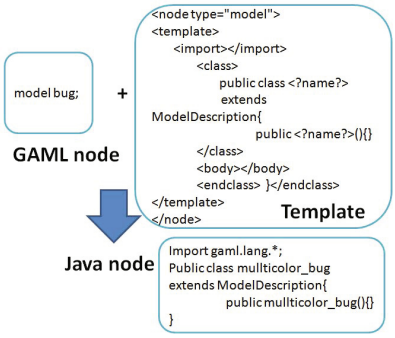

Fig. 10. Case Model

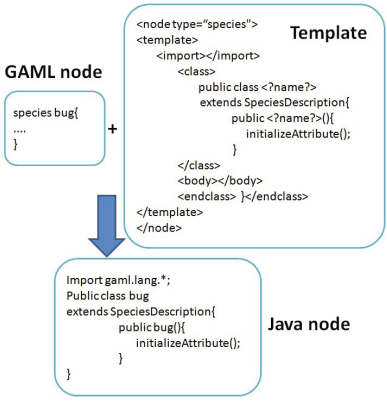

Fig. 11. Case Species

Convert Attributes, Behaviors. Almost operators, type, statements, and even skills, are interpreted thanks to annotation, and can execute thanks to GAML additions helpers. The helpers is an Java interface linked to Java class which take care on process the command. The idea is focus to declared annotations, implement a Java class to read all the annotation, and then translate it to Java syntax. When GAML model compiled, by mixing the tree and Java syntax, we have translated whole model into Java class.

Attributes declaration syntax contains two part: type and name, which are converted into declaration of variable in Java. In contructor of species, there is an initialization of these attributes, with default values.

Behaviors, which are Statements (simple and complex), is converted into Java's methods. These methods are called in Species body at each simulation step.

Operators include unary, binary or nnary, as example, binary operator 'OR' take two parameters, left_operand and right_operand. Its description in annotation (Fig. 12) will be translate in form of method which have constructor in form *_keyword(param1, param2,...)* , expression in GAML, we have the annotation in original Java class.

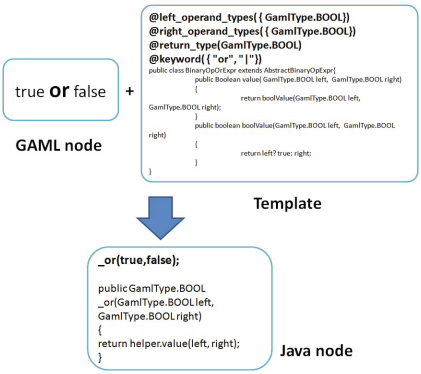

Fig. 12. Case Operators

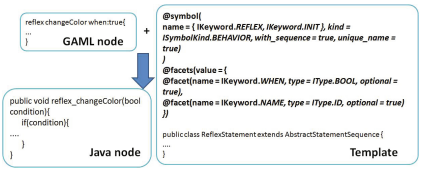

Fig. 13. Case Statements

With this annotation processing method, we can translate automatically almost GAML syntax. And it's so flexible when the grammar change by annotation. The name, the parameters, return types, will be updated automatically by annotation. But it still has some complex statement and skill which face the problem of logical and syntactic in Java, that can be solve by merging the two strategies, using both template and annotation.

5.3 Results of Dynamic Compilation

By using this model, we execute simulation 100 times and get the average statistic about running time, memory using. Regarding the advantages of compiling dynamically species into Java, we can see the following:

- Faster execution (although we can't prove it now, it is fairly obvious that compiled code should be faster than interpreted one, especially because Java code can be compiled on the fly to native code by the JVM).
- Better verification (for the code to compile properly, it needs to be correct in GAML).
- Possibility to further optimize the execution of agents by changing the Java code directly (instead of being restricted by the GAML set of primitives).

Scenario	Number of agents	Compile Time (s)		Running resources (Mb RAM)		Execution time for 400 simulation steps (s)	
		Original	Java	Original	Java	Original	Java
1	100	4	3	225	226	5	8
2	10 000	56	43	310	286	5	9
3	100 000	149	126	531	403	6	10

Fig. 14. Compare between dynamic compilation with original compilation method

6 Conclusion

This paper has established an operational model from conceptual model for a simulation platform, applicant to GAMA platform. With algorithm to traversing AST, combining templates and annotation, compiler create an operational model correspond with conceptual structure. With this research, model in GAML was convert into Java syntax, containing species, theirs attributes and simple reflex behaviors . This method has implemented into GAMA in form a plugin to facilitate the process of converting. In the next time, we will made entire structure of model with complex behaviors, skills and linking between multi model.

Acknowledgments. This publication has been made possible through support provided by the IRD-DSF.

References

1. Auslander, J., Philipose, M., Chambers, C., Eggers, S.J., Bershad, B.N.: Fast, effective dynamic compilation. In: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation, PLDI 1996, pp. 149–159. ACM, New York (1996)
2. Bebenita, M., Brandner, F., Fhndrich, M., Logozzo, F., Schulte, W., Tillmann, N., Venter, H.: Spur: a trace-based jit compiler for cil. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 2010, pp. 708–725. ACM, Reno/Tahoe, Nevada (2010)
3. Benson, T.: UML and XML. Principles of Health Interoperability HL7 and SNOMED. Health Information Technology Standards, pp. 51–70. Springer, London (2012)
4. Bettini, L.: A DSL for writing type systems for xtext languages. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, pp. 31–40. ACM, New York (2011)
5. Bryant, J.: Java syntax. In: Java 7 for Absolute Beginners, pp. 15–33. Apress (January 2011)

6. Chambers, C., Ungar, D., Lee, E.: An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. SIGPLAN Not. **24**(10), 49–70 (1989)
7. Gong, J., Cheng, R., Cheung, D.W.: Efficient management of uncertainty in XML schema matching. The VLDB Journal **21**(3), 385–409 (2012)
8. Hassanieh, H., Indyk, P., Katabi, D., Price, E.: Simple and practical algorithm for sparse fourier transform. In: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, pp. 1183–1194. SIAM, Kyoto (2012)
9. Henderson-Sellers, B.: Modelling languages. On the Mathematics of Modelling. Metamodelling, Ontologies and Modelling Languages. SpringerBriefs in Computer Science, pp. 63–74. Springer, Heidelberg (2012)
10. Ishizaki, K., Ogasawara, T., Castanos, J., Nagpurkar, P., Edelsohn, D., Nakatani, T.: Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs. SIGPLAN Not. **47**(7), 169–180 (2012)
11. Jones, J.: Abstract syntax tree implementation idioms. In: Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003) (2003)
12. Kerr, A., Diamos, G., Yalamanchili, S.: Dynamic compilation of data-parallel kernels for vector processors. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO 2012, pp. 23–32. ACM, New York (2012)
13. Koju, T., Tong, X., Sheikh, A.I., Ohara, M., Nakatani, T.: Optimizing indirect branches in a system-level dynamic binary translator. In: Proceedings of the 5th Annual International Systems and Storage Conference, SYSTOR 2012, pp. 5:1–5:12. ACM, New York (2012)
14. Kulkarni, P., Arnold, M., Hind, M.: Dynamic compilation: the benefits of early investing. In: Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, pp. 94–104. ACM, New York (2007)
15. Liu, Y., Fong, A.S.: Heuristic optimisation algorithm for java dynamic compilation. **6**(4):307–312 (2012)
16. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. **37**(4):316–344 (December 2005)
17. Parr, T.J., Quong, R.W.: ANTLR: a predicated-LL(k) parser generator. Software Practice and Experience **25**, 789–810 (1994)
18. Psaila, G.: On the problem of coupling java algorithms and XML parsers (invited paper). In: 17th International Workshop on Database and Expert Systems Applications, 2006. DEXA 2006, pp. 487–491 (2006)
19. Rohl, M., Uhrmacher, A.M.: Flexible integration of XML into modeling and simulation systems. In: Proceedings of the 37th Conference on Winter Simulation, WSC 2005, pp. 1813–1820. Winter Simulation Conference, Orlando (2005)
20. Rohl, M., Uhrmacher, A.M.: Composing simulations from XML-specified model components. In Proceedings of the 38th Conference on Winter Simulation, WSC 2006, pp. 1083–1090. Winter Simulation Conference, Monterey (2006)
21. Spafford, K.L., Vetter, J.S.: Aspen: a domain specific language for performance modeling. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 84:1–84:11. IEEE Computer Society Press, Los Alamitos (2012)

22. Taillandier, P., Vo, D.-A., Amouroux, E., Drogoul, A.: GAMA: a simulation platform that integrates geographical information data, agent-based modeling and multi-scale control. In: Desai, N., Liu, A., Winikoff, M. (eds.) Principles and Practice of Multi-Agent Systems. LNCS, vol. 7057, pp. 242–258. Springer, Heidelberg (2012)
23. Tam, D., Wu, J.: Using hardware counters to improve dynamic compilation. Technical report (2003)
24. Touraille, L., Traore, M.K., Hill, D.R.C.: A mark-up language for the storage, retrieval, sharing and interoperability of DEVS models. In: Proceedings of the 2009 Spring Simulation Multiconference, SpringSim 2009, pp. 163:1–163:6. Society for Computer Simulation International, San Diego (2009)
25. Ueberhuber, C.W.: Scientific modeling. Numerical Computation 1, pp. 1–8. Springer, Heidelberg (1997)
26. Vo, D.-A., Drogoul, A., Zucker, J.-D., Ho, T.-V.: A modelling language to represent and specify emerging structures in agent-based model. Principles and Practice of Multi-Agent Systems. LNCS, vol. 7057, pp. 212–227. Springer, Heidelberg (2012)