Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Tomáš Fechtner

MPS-based Domain-specific Language for Defining RTSJ Systems

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Michal Malohlava
Study programme: Computer Science
Specialization: Software systems

Prague year 2012

I would like to thank my supervisor Michal Malohlava for his advices, observations and help with thesis text. Further I would like to thank the developers of MPS especially to Alex Shatalin for their support during development.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... Tomáš Fechtner

Název práce: MPS-based Domain-specific Language for Defining RTSJ Systems

Autor: Tomáš Fechtner

Katedra: Katedra spolehlivých a distribuovaných systémů

Vedoucí diplomové práce: RNDr. Michal Malohlava

Abstrakt: Real-time Specification of Java (RTSJ) je rozšíření pro jazyk Java, které umožňuje využít tento jazyk pro tvorbu real-time systémů. Nicméně náročnost použití a složitý programovací model RTSJ s její manuální správou paměti vede často k chybám. Pro ulehčení vývoje RTSJ systému by proto bylo přínosné poskytnout rozšíření jazyka Java pomocí konceptu domain-specific language (DSL). Cílem tohoto DSL by bylo umožnit bezpečnější a intuitivnější vývoj. K tomu je však potřeba najít kompromis mezi všestranností tohoto řešení a jeho použitelností pro uživatele. Tyto požadavky však jdou často proti sobě. Jednou z možností pro vytvoření DSL je použít Meta-Programming System (MPS). Tento systém umožňuje vyvíjet nové doménově specifické jazyky a příslušné projekční editory poskytující rozdílné možnosti jak spravovat kód. Tato práce vytvoří DSL a příslušný generátor kódu umožňující vývoj RTSJ systémů, to vše pomocí MPS platformy. Dále pak provede zhodnocení takto vytvořeného DSL pomocí jednoduché studie. Nakonec tato práce posoudí vhodnost MPS jako platformy pro vytváření DSL.

Klíčová slova: rtsj, dsl, mps, real-time, java

Title: MPS-based Domain-specific Language for Defining RTSJ Systems

Author: Tomáš Fechtner

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Michal Malohlava

Abstract: The Real-time Specification of Java (RTSJ) is an intention to introduce Java as a language for developing real-time system. However, the complexity of their development and a non-trivial programming model of RTSJ with its manual memory management often lead to programming errors. To mitigate the development of RTSJ systems it would be beneficial to provide an internal domain-specific language (DSL) extending the Java language which would allow to develop the systems in more intuitive and safer way. However, it is needed to find compromise between solution's power and level of usability, because this two attributes go often against each other. One possible way of DSLs creation concerns the Meta-Programming System (MPS). It allows to develop new domain-specific languages and corresponding projectional editors enabling different views on code. This thesis proposes a design and implementation of the DSL on the top of the MPS platform and corresponding code generator enabling development of RTSJ systems. Furthermore, the thesis provides a simple case-study to evaluate a proposed DSL. Additionally, the thesis assesses the suitability of MPS as a DSL-development platform.

Keywords: rtsj, dsl, mps, real-time, java

Contents

Introduction and Motivation

Nowadays, automatization is the key concept which is adopted by current systems, from relatively simple ordinary washing machine at home to rather complex systems like production lines in factories or control systems for airplanes. These processes are always managed by an information system that receives data about the surroundings and reacts accordingly. Awareness of the current situation is mediated by various sensors measuring temperature, motion or time. Responses are then performed according to purely deterministic behavior configured by developers.

Tasks that controls these systems, work with the real world and therefore timing of their jobs is crucial. There must be defined scheduling properties for each tasks, moreover behavior for overrunning these deadlines should be specified. Besides of restrictions for time-consumption, there is a necessity for complex tasks' scheduling with various priorities. Systems, which fulfill these demands, are called real-time systems.

In the 1980s the most widespread language was the C language and therefore also the most of real-time systems were written in it or in the Ada language, which was originally planned mainly for these systems.

In the second half of the 1990s new languages have been developed, providing higher levels of abstraction, such as Java and C#. These languages, in contrast to the C language, shields the user from the low-level tasks such as allocation and deallocation of memory, calculation with pointers or typecasting variables to any types. Thanks to this simplification software development has become easier. Furthermore, errors associated with memory (like memory leaks, access to memory that the process is not owner etc.) have disappeared. Faster development with less number of errors is obviously cheaper, which is very important for commercial sphere.

These benefits are not for free. Because the allocation and deallocation are not directly written in code, there muse be some process, which manages these operation at run-time. In standard way it is solved by the `garbage collector`. It is launched from time to time and detects, which variables are inaccessible and deallocates them. This job can lasts for relatively long time and is started in unpredictable moments. Which is obviously a problem if we want to use such language for programming a real-time system.

Because the advantages brought by this new generation of languages are so significant, there has been designed a special Java's specification exclusively for real-time systems, so called real-time specification for Java (RTSJ). The specification provides a new class of threads, which is able to define more precisely their run time and priority. There are also introduced new types of memories besides of heap memory - immortal memory and scoped memory. The entire specification has many new rules, e.g., the variable stored in the immortal memory can not refer to scoped memory. These changes enable that threads using new types of memories do not need the garbage collector and can safely interrupt it. Thanks that full real-time system using RTSJ can be developed.

These new features are quite sophisticated to be used in a proper way, especially utilization new types of memory is not trivial. It would therefore be

appropriate to create a framework, which work would involve both the automatic setting and encapsulation of often repeated pieces of code and checking mentioned rules. Recently concept of domain-specific language is widely used during system development, so it would be nice to use this principle instead of implementing ad-hoc libraries or frameworks.

A domain-specific language (DSL) is not innovation coming from recent years, nowadays DSL is used much more thanks to the development of domain-specific modeling. It is a language that is designed for a specific problem or domain. Its purpose is to simplify application development for developers by introducing domain-specific concepts and vocabulary [3]. DSL is usually represented by a grammar describing domain concepts. Corresponding concrete syntax can be textual, graphical or a combination of both methods. For example, the user draws a graphical state machine diagram and enters the action for each state in the text form. Then associated generator produces classes, listeners and other necessary stuff, which would otherwise the user has to write himself in a programming language. Typically the program created by DSL's editor is not directly translated into binary code, but the generator generates a solution in a general programming language and then an appropriate compiler is launched on this generated code. Hence the creator of DSL does not have to write a compiler.

To create our own DSL we have chosen JetBrains MPS as IDE. It is a comprehensive integrated development environment meeting the modern requirements. It supports simple creation of all DSL components - structure concepts, their projection, behavior and interactions, data flow and corresponding code generator. In addition, there is a possibility to directly write a program using newly designed DSL. JetBrains MPS is still being improved and developers are adding new features. The project itself is open sourced under the Apache License 2.0.

Goals

The overall goal of this thesis is to create a DSL for RTSJ as an extension of Java. The DSL should allow developers, famous in Java, to develop a real-time system in a easier way than through learning the RTSJ. We will stress on easy usage and coverage of all important RTSJ's constructs, which are needed for creating a functional real-time system. The entire project will be created using JetBrains MPS. An evaluation of this tool will be a part of this thesis, or any suggestions for the developers how to improve this product.

Structure of Text

This thesis starts with description of the background, which is required to perform a problem analysis. First RealTime Specifications for Java is depicted in *Chapter 1*. *Chapter 2* acquaints with JetBrains MPS. *Chapter 3* revisits goals of the thesis. After this introduction part there are chapters dedicated to design of the DSL, in *Chapter 4* the analysis of language takes place and it provides top-level decisions about the language. *Chapter 5* contains a description of implementation. The case study based on the created DSL is placed in *Chapter 6*. Evaluation of DSL based on example from the previous chapter and evaluation of JetBrains MPS

is described in *Chapter 7*. *Chapter 8* presents related work. The last *Chapter 9* contains conclusion and possible challenges for future work.

1. RTSJ

1.1 Reason of Creation

The Java programming language is not suitable for development of real-time systems. The first obstacle, which most of programmers certainly realize, is a *garbage collector*. The garbage collector is automatic memory management, which checks each object in program, if it is still accessible. That means, if there still exists a reference to that object. Its non-deterministic launching and non-trivial processor's time consumption are for programming of real-time systems unusable. The garbage collector is not in Java specification, but without it, there must be used another tool for memory management.

The second trouble is linked to a lack of variability in a specification of thread's priority and scheduling. In real-time systems there are often tasks with periodical execution. Their job mostly consists of checking their value and executing some actions relating on this value. And this activity must be done regularly. The another typical task is, that some code has to be launched before a deadline aperiodically. The programmer needs a tool to define these special behavior, but in pure Java it is not possible.

Java community had to solved these two main groups of problems - memory management and threads scheduling. History of *real-time specification for Java* (RTSJ) begun in march 1999. On January 2002 RTSJ 1.0 was accepted by Java Community Process - organization for developing standard technical specifications for Java technology. Development continued by publishing revision of specification 1.0.1 in spring 2005. In Spring 2006 working on RTSJ 1.1 as JSR 282 has been started. Today RTSJ is still in a development stage and the actual revision is 1.0.2 since Summer 2006. This thesis describes and uses the last released revision 1.0.2. [6, 7].

1.2 Basic about RTSJ

RTSJ specifies new classes, which extend the original Java language. The main differences compared to Java are:

- real-time threads with special behaviors, asynchronously event handlers and scheduling;
- new memory management including non-heap memory, restrictions in referencing and possibility to allocate/deallocate limited area of memory;
- utilities for better work with these new types of memory and threads, furthermore tools necessary for real-time systems like access to raw memory.

Besides of these new elements of language, the world of real-time systems has its own design patterns like wedge threads or solutions of communication between threads [2, 16, 22].

RTSJ only specifies declaration and behavior of classes and methods. Several implementations of RTSJ exists, both from commercial and academic spheres

e.g., from Timesys, IBM or Oracle. In domain-specific language, created in this theses, is used implementation from Timesys [19].

Program using classes from RTSJ needs a special virtual machine (VM), which can handle extended variability of thread's priority and other added functionalities. There are many VMs, each implements different subset of RTSJ [17].

1.3 Threads

Class `RealtimeThread` extends class `Thread` from original Java. Constructor has several new parameters, which define new settings and behavior. New methods are mostly setters for these new parameters, so programmer has possibility to decide between defining these settings by parameters in constructor or by these setter-methods. We would like to describe the most important classes used as parameters in the constructor and their variability.

- `SchedulingParameters`, respectively its subclass `PriorityParameters`, is purposed to set a priority of thread. Threads can have maximally 32 priorities.
- Class `ReleaseParametres` serves for specification of scheduling behavior of threads. It has 3 subclasses `PeriodicParameters`, `AperiodicParameters` and `SporadicParameters` (subclass of `AperiodicParameters`). `PeriodicParameters` is used for periodic tasks with defined period. Job of `AperiodicParameters` may be launched at any time before deadline. `SporadicParameters` is intended for not regular launching as it is in `PeriodicParameters`, only with minimal interval between launching. Besides of described parameters, each subclass has miss-handler and overrun-handler. They serve for treat exceptional states, when thread overruns cost time respectively deadline time.
- The last interesting parameter is `MemoryArea`, that define in which memory area thread will run.

`NoHeapRealtimeThred` is a specialization of `RealtimeThread`. As the name indicates, it is a thread, which can neither allocate nor refer variables on heap. Due these limitations `NoHeapRealtimeThread` can safely interrupt the garbage collector at any time. This thread must run in immortal or scoped memory.

With threads another problem relates - the priority inversion. Let's consider two threads, High priority (H) and Low priority (L). The L starts and locks object X, after then the H starts and tries to access to X, which is locked. H must wait for L, until it unlocks X. That is not ideal but correct. But consider another thread Medium priority (M), which starts after L locking X. M has higher priority than L and does not need X, therefore the scheduler allows M working without interrupting. That is the priority inversion, because H has to wait for M, which has lower priority. JCR has evaluated algorithm for `synchronized` keyword as not sufficient to fulfill both preventing the priority inversion and preempting the garbage collector in each situation. Therefore RTSJ provides wait-free queue classes to solve this problem, names of these classes are `WaitFreeReadQueue` and `WaitFreeWriteQueue`.

1.4 Memory

RTSJ need some part of memory, which is not managed by the garbage collector. Because threads working with variables placed only in these parts of memory can safely interrupt the garbage collector. Thanks to this property programmer does not worry about random delay from the garbage collector and the whole real-time system can run in a deterministic way. But if these parts of memory are not managed by the garbage collector there must be another system for handling deallocation.

In RTSJ there are two new types of memory:

- *Immortal memory* - variables in this memory are immortal. It means, that they have never been unallocated. There is only one Immortal memory in each program and it is shared among all threads.
- *Scoped memory* - area of memory with limited life. Programmer can create scoped memory at any time. Variables in this memory are not deallocated by the garbage collector. But when each variable in this memory is not accessible, memory is completely deallocated.

Scoped memory is divided further into two implementation - `LTMemory` and `VTMemory`. LT means linear time of allocation new objects in this memory, therefore time of creating a new object is predictable. `LTMemory` never launches the garbage collector on its objects. But calling `new` will fail, if the area of memory is full. Unlike `LTMemory`, `VTMemory` could spent unlimited time during allocating new object. That allows specific RTSJ implementation to use a memory management. `VTMemory` can even implement some kind of garbage collector, which could be faster thanks to restrictions of referencing in RTSJ memories. However the RTSJ does not specify any benefits of `VTMemory`, so implementation could declare `VTMemory` such a simple wrapper over `LTMemory`.

Besides of these new memories, the programmer can still use heap memory and local variables. But he has to remember limitations caused by the garbage collector and utilize heap memory only for such cases like initialization during start of application.

There are some difficulties related to utilizing of scoped memory. The first issue is, that scoped memory needs to know its size when is created. To be precise constructor's parameters are initial and maximum size of memory. RTSJ provides class `SizeEstimator` mitigating object size estimation. `SizeEstimator` has methods, which expect types as parameters and calculate their sizes.

The second problem, which makes using scoped memory much more difficult, is, that variables in immortal and heap memory are not allowed to reference variables in scoped memory. Furthermore, variables in scoped memory can reference only variables from the same scoped memory or an outer ones. These rules are needed to secure, that scoped memory does not become immortal.

The last thing connected with memory, which we would like to mention, is access to the raw memory. In real-time systems there are typically some measuring devices and application needs to access them. This is done typically by an access to their raw memory. RTSJ specifies class `RawMemoryAccess`, which methods allows the access by setting offset of memory and expected data type.

2. JetBrains MPS

2.1 Basic Info

JetBrains is a company focused on creating of integrated development environments (IDE). JetBrains MPS (Meta Programming System) enables easy creation of language constructs. MPS is used both for creating DSLs and for using these DSLs [10].

JetBrains MPS started in 2003 as an research project. JetBrains company has used this IDE to develop their own products since 2006. MPS is distributed as a open-source product under Apache 2.0 license.

MPS is not a classic IDE for creating DSL. It is not grammar-based tool, where creating of DSL consists of defining rules for individual tokens and lexemes as does Flex[1] and Bison[2]. MPS belongs to the so-called *Language Workbenches*. Tools, which enable language oriented programming with a projectional editor in persistent abstract representation [4, 20].

The basic idea of MPS is to extend an existed *base language* or some other DSL created in MPS. The base language consists of *concepts*. The concept is an element of language in MPS environment, it describes how the elements look like, behave, are generated etc. These individual aspects of concept are defined in *models*. Each model is responsible for clearly defined part of DSL declaration. The default base language in MPS covers the whole Java language and translates its statements, expressions, assignments etc. to elements, with which both user and IDE can work.

By this extending the user can define new concepts for base language or he can extend the existing one. That means, that he creates some new statements, behavior or whatever else he wants to add to Java language. If there was only this possibility, it would be very restrictive. But user does not have to extend the default MPS's base language, he can build on basic concepts like `BaseConcept`, `Attribute` or `DataTypeDeclaration`. With these tools he can create for example a small mark-up language generating HTML/XML. Or he can create own base language based on another language than Java. Furthermore, an extensible C language from Markus Völter and his colleagues exists [21].

2.2 Description of IDE

The main difference from other IDEs, such Eclipse or Microsoft Visual Studio, is, that MPS's structure of code is strictly tree-like. Even in the editor the user sees and edits a tree-like structure. The appearance is similar to another IDEs, but the user identifies difference during editing. Let there is a simple assignment and the user wants to insert it into an if-statement as a body. The user cannot simply write braces before and after this assignment. Because this assignment is a AST[3] node, he must create a new node, the if-statement, and has to set the assignment

[1] http://flex.sourceforge.net/

[2] http://www.gnu.org/software/bison/

[3] Abstract syntax tree

as a child of if-statement. Fortunately, IDE provides user tools for easier job, e.g., the user can utilize context menu[4] and surround selected code with common statements like if, while, try etc. On the other hand, the tree-like structure produces benefits like fast data-flow analysis, inherent AST transformation or easy language embedding.

The MPS itself can be divided in two parts:

- language editor, where the user defines a DSL, its structure, behavior, generator etc.;
- solution editor, where anybody can use the defined DSLs and use them to develop own real-time systems. Results of this editor are called *solution*.

2.3 Language Editor

A language consists from concepts. Each concept can have a definition in one or more *aspects* of language. These aspects are structure, editor, generator etc. User can see concepts aggregated by aspects or can choose one concept and browse through its definitions in all aspects.

Now we would like to name the most important aspects and briefly describe their functions and possibilities.

2.3.1 Structure

The structure is the most important aspect. Each concept must define this aspect. The concept is exactly a new type of node and in this aspect user defines its name and on which basic concept it is based. Furthermore, there is defined from which other concepts is composed:

- child - instances of other concepts with role and required cardinality;
- reference - references to already existed instances of concepts with role and required cardinality;
- properties - only primitive types and enumerations.

Role in children and references are similar to name of variable in classical programming. In the UML language children are called composition and references are called aggregation.

2.3.2 Editor

The Aspect Editor serves for definition concrete syntax - i.e., how the concept will look like in solution editor. The user sets positions of all children, references and properties here. There is a possibility to insert static texts, for example, as a description of concept's elements. The user can style each element, can even add condition for displaying. The IDE provides the user an easy way how to set mutual positions of elements including their indentation.

[4]shortcut key CTRL + ALT + T

2.3.3 Generator

To be precise there are two kinds of generators. The first one provides transformation from one concept to another. This is typically used, when the user extends the base language. For example the user adds a concept, which removes boilerplate code. So generator replaces this new concept by generating necessary code for proper functioning and this generated code consists from concepts, too. In the base language there are already concepts for each item of programming language as assignment, class, expression etc.

The second generator is on lower-level than the transformation generator. It generates directly text. The transformation generator is launched typically as a first one, the text generator is launched afterward. If the user extends the base language he usually does not need to use text generator. But creating a new base language requires it.

Description of transformation generator

In DSL created in this theses, the transformation generator (further only generator) will be used mainly, therefore it will be described more deeply. A main file of the generator is *mapping configuration*. In this structure the user defines all transformations, which is possible to place there, or for better clearness in special files referenced from mapping configuration. Mapping configuration has several options to parametrize its behavior and then several groups of *rules*. Rules serve to define a way of concepts' generation (e.g. from which is result generated, possible condition of generation, which *template declaration* is used). Each group, respectively type of rules, is designed for specific kind of usage.

A template declaration is a file, which describes how the result of generation will look. In the template declaration there could be utilized *macros*, which change final result according to parameters of currently processed concept's instance. There are three types of macros:

- *property macro* - substitutes value in template by property of input node;
- *reference macro* - substitutes reference in template by reference to input node or its fields;
- *node macros* - group of macros with the largest changes to template. They can change even the whole nodes in template. For example, the macro `$COPY_SRC$` substitutes the node in template by the input node or its fields. The macro `$LOOP$` is used repeatedly for generating nodes. The macro `$IF$` generates an output in dependency on condition.

There are more files/elements besides of templates and mapping configuration in generator aspect. A template switch could be created for better clearness. It serves as a normal switch, according to the condition it calls a specific template declaration.

The template declaration must be instance of a selected concept. It could not be a text string. Sometimes it is necessary to use variables or objects, which declaration and initialization are not required to be generated. In that case, the generated code is marked as a *template fragment* and the rest of code has only a supporting role.

We already mentioned rules, now let us examine them in detail. Rules define basic definitions for each generation. We will describe types of rules gradually:

- *Conditional root rules* - on condition, set existed instance as new root node. This instance must be an instance of the base language concept in generator aspect;
- *Root mapping rules* - on condition, transform instance of concept to root node by using template declaration;
- *Weaving rules* - modify existing node by adding a new child to it. This already existed node was created by mapping/reduction rule;
- *Reduction rules* - on condition, transform instance of concept by using template declaration, the result is not root;
- *Abandon rules* - on condition, ignore specific instances of concept;
- Scripts - pre and post processing scripts for some initialization.

Example of generation

Because the generator is rather complicated, we will present a simple example. We will generate statement `basicFor`. That concept will behave like a specialized for-statement for incrementing from zero to user's maximum value. The required concept's structure is depicted in Figure 2.1.

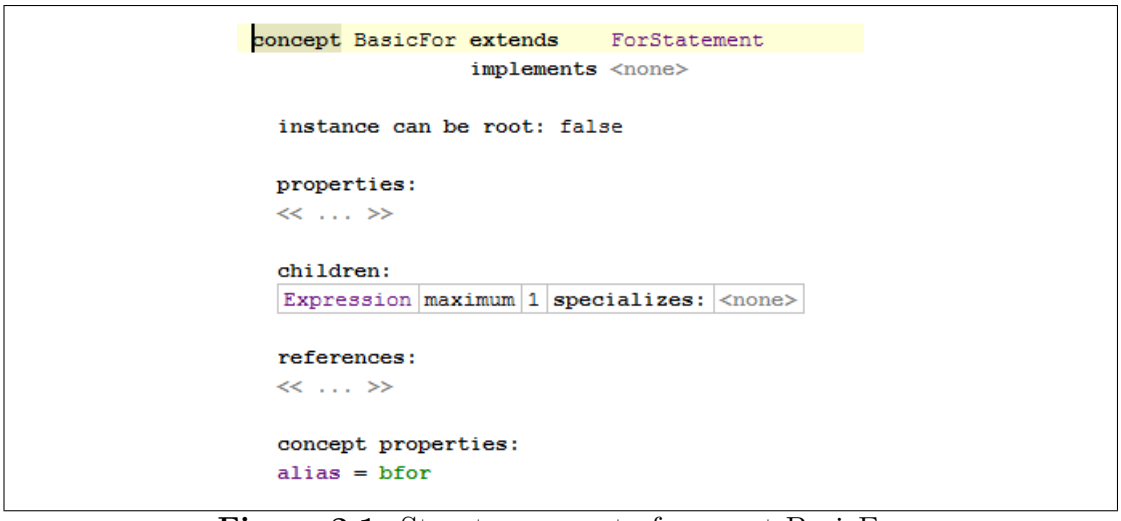

Figure 2.1: Structure aspect of concept BasicFor

Then we must create a new reduction rule in mapping configuration, which will transform each instance of our new concept `BasicFor` to for-statement of the base language. This rule should be done always, thereby no restrictive condition is needed. The result is in Figure 2.2.

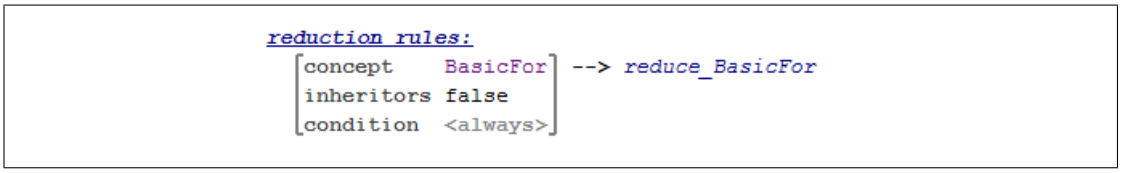

Figure 2.2: Mapping rule for concept BasicFor

The last needed action is the creation of template declaration. As we can see in Figure 2.3, template declaration is an instance of concept, therefore whole code is a template fragment (surrounded by pink TF). There are two node macros `$COPY_SRC$`. The first one is used for maximum value and the second one for body. Definitions of macros are written in *Inspector*, special window used for extended definition, which is in the bottom of the same figure. There are one property and two reference macros, too. They serve for correct substitution of variable name.

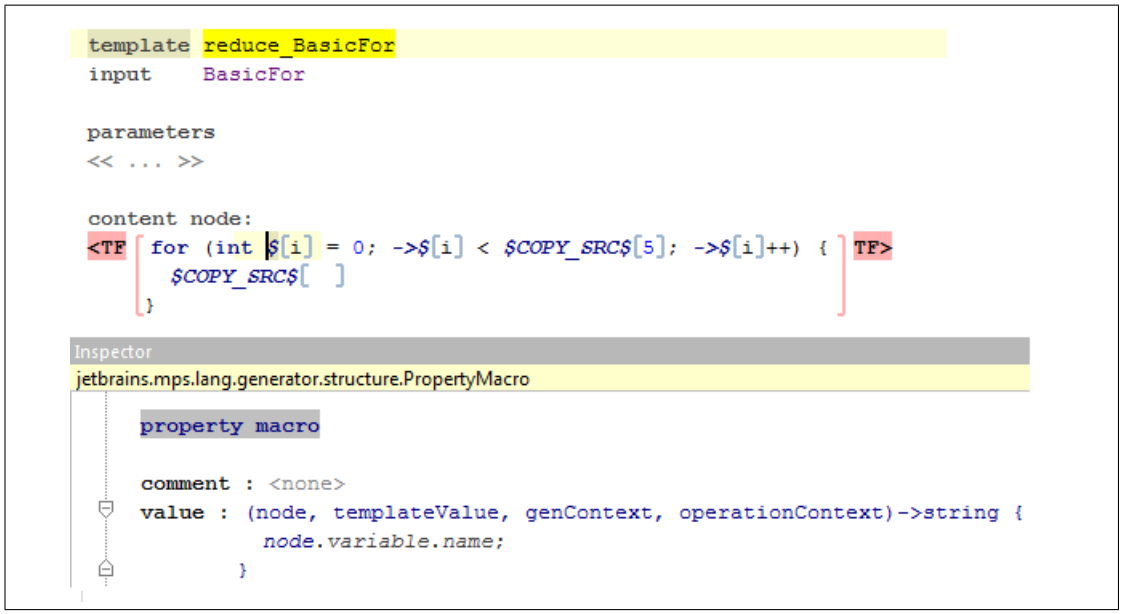

Figure 2.3: Template declaration for concept BasicFor

2.3.4 Data-flow

In Data-flow aspects the user can describe how should data-flow analyzer understand the new concepts. In MPS is already implemented analyzer for static analyzes of using variables (initializing and using) and potential null-pointer exceptions. The user can write own analyzer for analyzing own issues.

2.3.5 Other

- *intention - AST transformations* for more user-friendly working with concepts, e.g., transformations from one concept to another (e.g. `BasicFor` to `For`) or action, which wraps selected statements with the concept. They are triggered by the user from contextual menu only;
- *actions* - automatic actions after some user's activity. Contrary to intention, they are triggered by defined rules. For example inserting the equal sign after variable declaration causes creating initializer for this variable;
- *constraints* - restrictions for using concept, define which concept can be a child, a parent etc.;
- *behavior* - the user can implement methods, which can be called in other aspects;

- *type-system* - several types of checks such as subtyping, comparison or inference rules.

2.4 Solution Editor

Solution editor is normal IDE with tree-like structure as was described in Section 2.1 about basics of MPS. The user defines used DSLs and then he simply uses defined concepts. After finishing programming, he will build solution. That consists of code generating from concepts and then compiling it by a defined compiler. In case of this thesis, the building solution generates Java classes by definitions in *the aspect generator* of created DSL, then these classes are compiled by Java compiler.

3. Goals Revisited

In the previous Chapters 1 and 2 we have explained, why pure Java is not sufficient for real-time systems. We have presented the real-time specification for Java, its advantages, which make using Java for real-time system usable. On the other hand we have mentioned its disadvantages.

The thesis motivation is to create a solution, which makes programming in RTSJ easier. This solution will consist of designing a DSL, which will serve as layer on the top of RTSJ. This new DSL will be named as *RealTime Extension for Java* (RTEJ). RTEJ will introduce several new statements, which enable easy definition and utilization of RTSJ concepts especially connected with threads and memory areas. Furthermore, RTEJ will include concepts encapsulating selected design patterns. They will simplify specific use-cases and improve abstraction. The second task of RTEJ will be checking correctness of using memories and providing early-error detection, coming from incorrect usage using of new DSL concepts. Of course, besides of DSL, we will create a corresponding code generator, which will generate correct RTSJ code.

We would like to propose this DSL in the scope of Jetbrains MPS, therefore another goal of this thesis is to write an evaluation of the MPS. We will analyze experience with writing intended DSL, enumerate especially useful features, and try to suggest some improvements and mention errors or hard-to-use elements of IDE.

To evaluate designed DSL we will write a simple real-time system in RTEJ, to test the power of new language and evaluate a usability level. The example will cover most of features needed in standard real-time systems and will be executed on existed real-time virtual machine.

To summary, the main goals of the thesis are:

- to design and implement DSL for RTSJ and corresponding RTSJ code generator, which would provide easy-to-use way to implement real-time systems;
- to evaluate Jetbrains MPS, especially how MPS is efficient for creating DSLs and language extensions;
- to evaluate the DSL based on a case-study.

4. Design of Domain-specific Language

To fulfill the goals of the thesis, RTEJ must define new concepts, which cover necessary parts of RTSJ as was described in Chapter 1. For each concept, its structure, behavior and appearance should be analyzed. Further, the way of generation must be defined. Besides of concepts covering RTSJ, we have to design implementation of design patterns for real-time systems. Especially analysis of design patterns balances among user-friendly usage, difficulty of implementation and solution's power.

4.1 Basic Decisions about Language

RTEJ extends Java language by new statements in the form of MPS concepts. The new concepts can be divided into two groups. The first group represents concepts which behave like normal statements. The user uses them directly in Java code. The most of them extend concepts statement or expression, the user can write them on places expecting statements respectively expressions like normal elements of the language.

Using of concepts from the second group is unconventional. The user creates them like new class files. In solution viewer they can bee seen among other files. As was mentioned, the Jetbrains MPS is strictly tree-structured, so this property means that concepts from the second group can be root. Therefore we would name them as *root concepts*. In RTEJ root concepts is mostly used for various definition.

4.2 Code Concepts

Firstly, new elements of RTEJ language should be defined. In Chapter 1 we mentioned, that main changes concern threads and their scheduling and new memory types. As the first step we analyze threads.

4.2.1 Concepts Connected with Threads

We should enable the user to define real-time (RT) threads. There are three basic types of RT threads described in Section 1.3 about RTSJ. These types are defined by using specific subclass of `ReleaseParametres`. Each type has its own group of definition. We have two possibilities how to handle it:

- define one concept `RealtimeThread` with a switch, this switch would change required definition;
- create three concepts derived from abstract `RealtimeThread`, each concept would have only specific definition required by this type, no switch, no changes.

One common concept with switch has an advantage in an easier way, when the user changes his mind and wants to have other type of RT thread. Solution with three concepts has on the contrary an advantage in stricter type checking and the user has to analyze architecture of new RT system deeper. For better type-checking we have chosen for RTEJ the solution with special concept for each type of RT thread.

Another decision is, whether concepts for defining threads could be used as root or not. Typically it is not a problem, whether launching and initializing of RT system take long time. Much more important is, that during the run of system, there are no delays and increasing CPU requirements due to starting other objects including new threads. So we have decided, that preferred time to define threads should be before the moment, when the system is launched.

Because RT threads should be defined before start of application, they can be defined as root concepts. That style is clear, the user has possibility to have all RT threads' definition grouped in one place, e.g., in a special package. However, there can rise a demand for allocation of count of RT threads, which the user does not know before launching of application. These RT threads must be allocated dynamically, therefore RTEJ should provide an opportunity to allocate and run RT thread even during runtime. Hence RTEJ has two ways of allocating with threads:

- root-concept threads with unique names, which are used in other concepts, e.g. modes;
- anonymous in-code defined threads used only for dynamically starting.

There are two ways in RTSJ, how to insert user's logic into threads

1. implement a new class, which extends class `RealtimeThread`;
2. refer an instance of class with logic to `RealtimeThread`'s constructor, this class has to implement interface `Runnable`.

The second way is better for purposes of this thesis. The main reason is, that the user should not be forced to know how to extend `RealtimeThread` class. RTEJ demands the instance of class implementing logic of thread in a concept for the RT thread definition. The arrangement enables initializing class, even with parameters, before referring to thread. E.g. by parameters in constructor or using setter-methods. The final looks of thread is presented in Figure4.1.

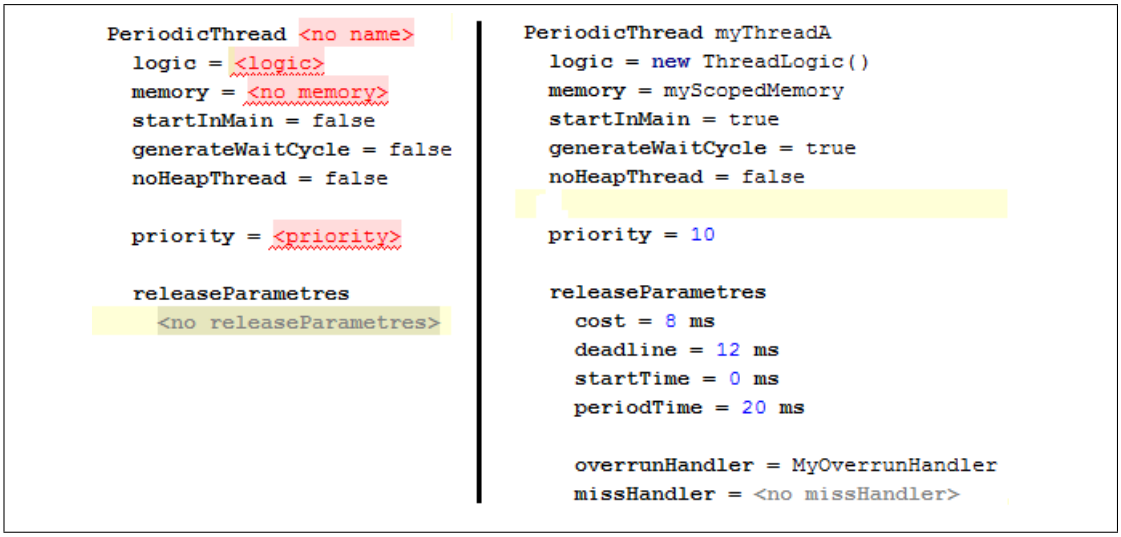

Figure 4.1: Looks of periodic thread in solution.

The user needs, besides of defining RT threads, the possibility to run and interrupt them. To realize that, RTEJ provides two simply concepts `runThread` and `interruptThread`. The user is able to use them as normal statements in code with only one parameter - name of thread. Both of them are presented in Figure 4.2.

```
runThread ( myThreadA ) ;

interruptThread ( myThreadA ) ;
```

Figure 4.2: Looks of concepts for handling with threads.

System of periodic repetition of execution threads logic is in RTSJ typically performed with calling method `waitForNextPeriod`. Standard code looks like do-while cycle with calling the mentioned method as a condition. Body of this cycle is then a piece of code, which should be executed periodically. An example is attached in Listing 4.1. The simplest way is to create a concept `waitForNextPeriod` as an expression. With that feature the user can wrap the logic of thread with this do-while cycle on his own. But that simple solution would not remove boilerplate code, therefore RTEJ provides in thread definition a switch for auto generating the do-while cycle.

```
public void run() {
    RealtimeThread thisThread = currentRealtimeThread();
    do {
        // code, which is executed in each period

    } while (thisThread.waitForNextPeriod());
}
```

Listing 4.1: Example of using waitForNextPeriod

4.2.2 Concepts Connected with Memories

The definition of new scoped memory is much more simple than creation of a new RT thread. RTSJ requires only a type (LT or VT), initial size and maximum size.

On the other hand, memories are typically allocated dynamically according to the needs of application. Lets imagine a simple application - thread for calculation of measurement in each period. This calculation needs allocation of several variables, but they are needed only for one period. After that, the whole scoped memory with all variable is deallocated. Example is shown in Listings 4.2.

```
import javax.realtime.LTMemory;
import javax.realtime.RealtimeThread;
import javax.realtime.ScopedMemory;

public class MyThread extends RealtimeThread{
  private ScopedMemory scoped = null;

  private Runnable runInScoped = new Runnable() {
    public void run() {
      // ... calculation executed in local scoped memory ...
    }
  };

  public MyThread(){
    super();
    scoped = new LTMemory(256, 512);
  }

  public void run(){
    do {
      // ... code executed in default memory ....
      scoped.executeInArea(runInScoped);
      // ... code executed in default memory ....
    } while (RealtimeThread.waitForNextPeriod());
  }
}
```

Listing 4.2: Usage of local defined scope memory

To implement such a case, RTEJ must define not only anonymous memory area as it is in RT threads. These dynamic areas must be accessible from rest of class by name or reference. Moreover, the described class could be instantiated several-times and each instance needs its own memory area. This requirement has appeared during initial-implementation of case-study. Therefore RTEJ provides global and local memory areas. The *global area* is accessible from the whole application and exists only once. The *local area* can be used only inside the specific class and is instantiated for each instance of that class, naturally way of implementing is private attribute.

Besides of manner of access, the moment of allocation must be analyzed. Scoped memory can take a lot of memory, therefore there must be a way, how to postpone allocation to the moment of demands. From that reason RTEJ provides for both types of memory areas a technique how to allocate it immediately or later on. Lets recapitulate all possibilities for declaring memory area:

- global area - accessible from the whole application;
  - defined as a root concept - immediately allocation;
  - defined in code - allocation in the spot of allocation concept;

- local area - accessible only from its class, created for each instance of that class;
  - flag for immediate allocation set as true - immediately allocation in field declaration;
  - otherwise - allocation in the spot of allocation concept.

A local variable declaration brings a decision, how the user is able to reference that memory in other concepts. The first way is to create a new variable type and work with that memory areas as with normal variables. The second one is to manipulate with the local area as with the global one, that means with the name of concept and to let the generator to generate all needed stuff as correct references and definition of variables. To maintain the same style of referencing with the rest of application, referencing via name has been chosen.

Both concepts have been created in a similar way not to confuse the user. On the left side of Figure 4.3 is a layout of root concept declaration and on the right side is the declaration in code.

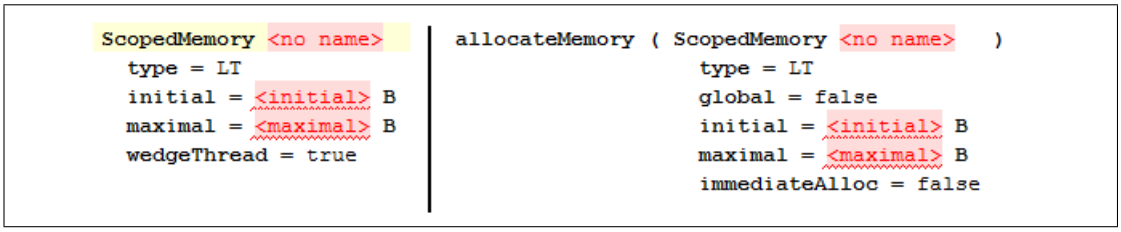

Figure 4.3: Appearance of declaring memory

Alongside of definition, the user needs tools for using memories. It should cover all important features from RTSJ

1. definition of real-time thread requires reference to main memory of that thread;
2. run piece of code in set memory;
3. allocating variable in set memory.

A definition of the main memory for thread is straightforward, the user simply writes the name into the definition of thread.

The second usage of memory could be implemented as a block statement, where statements in this block would be run in set memory. But generated code must be similar to Listings4.3, due to fulfill RTSJ demands. Repeating this piece of code would cause memory leaks by overhead from allocating still new anonymous `Runnable`, especially when this code is executed in immortal memory. Therefore RTEJ requires `Runnable` expression and user can save it in variable similar to Listings 4.2.

```
memoryArea.executeInArea(new Runnable() {
  public void run() {
    // code to execute
  }
});
```

Listing 4.3: Usage of local defined scope memory

The result concept is shown in Figure 4.4. On the top is unfilled concept and on the bottom of the figure is sample of inserting anonymous `Runnable` directly, but the user can insert arbitrary expression, too.

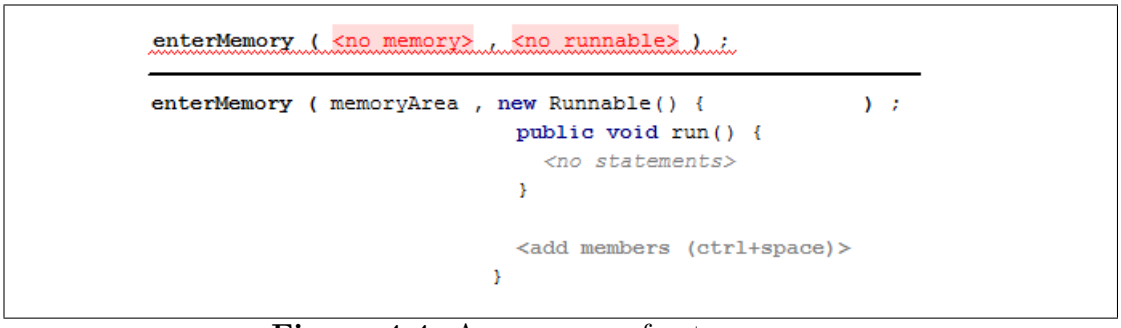

Figure 4.4: Appearance of enter memory

The last kind of memory utilization is a little bit complicated. Allocation of object in memory could be designed as an expression, which could be inserted to method's initializations or method's parameter etc. But this amount of freedom could tempt the user to use it as in normal Java code and rely on garbage collector, which means he would create unnecessary variables. Especially in immortal memory it would quickly fill up memory.

Due to this danger, RTEJ limits using of this allocating expression only as initializer in local variable declaration, field declaration and static field declaration. This allocation concept has two versions, one for allocating normal objects and the second one for arrays. Both these concept require a type of variable and the memory, where object should be allocated. In addition, array allocation needs a specification of its length. The appearance of both allocating expression is shown in Figure 4.5.

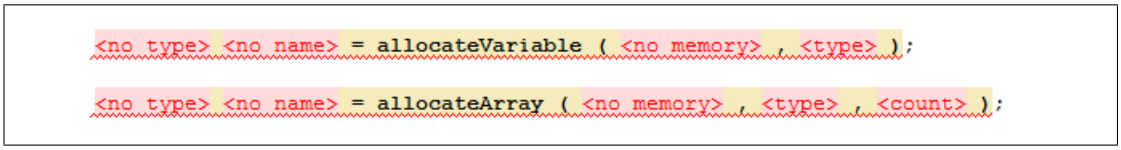

Figure 4.5: Appearance of allocating in memory

In the whole section we were speaking about the scoped memory. But each concept must work with the immortal memory, too. A correct solution is to use inheritance. Abstract concept called `MemoryArea` is created for this purpose. `ScopedMemory` and `ImmortalMemory` are its descendants. Unfortunately to proper function is needed to instantiate concept `ImmortalMemory` by user.

4.2.3 Helper Concepts

As we mentioned a little in Section 4.2.1 about RT thread, the user needs possibility to several initializations. The application's main method is generated by RTEJ generator and is filled by RTEJ threads and memories definitions. So the user needs another way how to set his initializing scripts. Concept `MainDefinition` serves for this. It is a root concept and provides possibility to set code to launch before and after generating initializations. This in-line code is referred as class implementing interface `Runnable`.

The initialization of the scoped memory needs the size of allocated memory in bytes. As was mentioned in Chapter 1, RTSJ defines class `SizeEstimator` for easier work, sample of usage is in Listing 4.4. RTEJ should provide concepts for using this class. Concepts in Jetbrains MPS are more like function call, than objects. So we must create a special concept for each user's action with size estimator.

1. The user must create a new estimator and name it. We define this concept like an allocation statement.

2. The user uses this estimator and sends a class to it, which he would like to estimate. As we said, no object-like solution is possible. So statement looks like a function with parameters: name of estimator, name and number of instances of class to estimate.

3. Finally concept for the method, which returns estimated size. It needs only the name of size estimator.

```
// inilization of SizeEstimator
SizeEstimator sizeEstimator = new SizeEstimator();
// reserve space for five instances of object
sizeEstimator.reserve(Object.class, 5);
// and space for two Doubles
sizeEstimator.reserve(Double.class, 2);
// find size of estimated types
int size = sizeEstimator.getEstimate();
// create scoped memory
LTMemory memory = new LTMemory(size, size);
```

Listing 4.4: Example of using SizeEstimator

An access to raw memory is similar to size estimator. To cover every important functionality of RTSJ, we must implement this class, too. Solution should be analogical to the size estimator. This time the three concepts are: initialization, get value of memory with sent offset and set memory with sent offset to sent value. However, initialization is unlike the size estimator root concept. Size estimator is used locally but access to raw memory is supposed to be well-defined and used global in whole app.

For better type checking, get and set concepts require the type of expected value. We restrict types to integer values of all Java sizes. The reason is, that raw memory access is used primarily to work with some sensors, which return numerical values. RTSJ offers a class for access to float values, too. We have decided to implement concepts for this class as pointless, because this solution would be completely the same as for integral values and not beneficial. But if potential users demand this possibility, it is not problem to add it.

All concepts for handling with size estimator are demonstrated in Figure 4.6.

```
SizeEstimator myEstimator = new SizeEstimator () ;

reserveInSizeEstimator ( myEstimator , AtomicInteger , 5 ) ;

Long size = estimateSizeEstimator ( myEstimator );
```

Figure 4.6: Appearance of hadnling with size estimator

4.3 Advanced Concepts

4.3.1 Computation Modes

alternation of phases or modes is very frequent In RT systems. E.g., RT system for plane control system would distinguish among modes launching, flying and landing. The user can write changes in system's controlling, which are inflicted by this new phase, directly into code. But much better is to have opportunity to define some modes and switch among them.

RTEJ offers a root concept Mode. The user is able to define to run and interrupt threads. He is able to change priorities of already running threads and even release parameters as is shown in Figure 4.7. Release parameters are represented as separate concept instead of just definition within thread's settings. That is, because these concepts are used on more places, in definition of RT thread and in changing parameters in Modes.

```
Mode myMode
  Run threads
  threadToStart

  Stop threads
  threadToStop

  Change priority
  threadA : 10

  Change release parametres
  threadB : cost = <no cost> ms
            deadline = <no deadline> ms
            startTime = 0 ms
            periodTime = 10 ms

            overrunHandler = <no overrunHandler>
            missHandler = <no missHandler>
```

Figure 4.7: Appearance of defining mode

After defining modes, the user can use their names directly in code in concept for mode change. This concept needs only the name of new mode to run. The user is able to define, which mode starts in initialization of RT system. This definition is done in concept main definition. Both usage of starting mode is presented in Figure 4.8.

```
changeMode ( myMode ) ;

Main definition
  before intilizing <no mainPreInit>
  after intilizing <no mainPostInit>
  starting mode myMode
```

Figure 4.8: Appearance of using modes in app

Using of modes is not demanded, modes are intended only for make the work easier. RT threads have a flag, whether they should be run immediately after

initialization of system. So it is only up to the user, which way he will choose for automatic starting of threads, either by selecting the starting flag or by starting mode with filled these threads as threads to run.

4.4 Patterns Concepts

In previous sections we analyzed and described concepts covering majority of RT-SJ. In addition we introduced our solution of often implemented system of modes, which could simplify the organization of system with many threads. Besides of modes there are many good practices, which are different from programming in non-RT systems. It would be nice, if RTEJ provides several of these real-time design patterns to the user as elements of language. They should be implemented as concepts, from which generator then generates an appropriate boilerplate code affected by user's parameters.

4.4.1 Wedge Thread

The scoped memory is deallocated automatically, when all objects are inaccessible. That is a good feature, e.g., thread is working in some scoped memory, then thread finishes its work and the whole scoped memory is correctly deallocated. So far so good, but what if the thread is periodical and the user wants to save some values for the next cycle? In this moment it is necessary to run another thread, which accesses the scoped memory and stays in memory until the original thread comes back. Naturally this support thread must not overload CPU. This pattern is called wedge thread.

We provide the user a flag in the definition of scoped memory. If the user sets this flag as true, a new wedge thread will be created. When the user needs to start wedge thread, he only uses our new statement `startWedgeThread`, which expects the name of memory as parameter. Why is required the name of memory instead of some name of wedge thread? Because the user does not need to know anything about auto-generated name of thread, he only wants to prevent the specific memory from deallocation. Once the original thread accesses the scoped memory again or holding of memory is not needed any more, the user uses the concept `stopWedgeThread`, again with the name of scoped memory. Defining flag in scoped memory was shown in last line of Figure 4.3, usage of starting and stopping concepts is presented in Figure 4.9. There is still worth mentioning, that wedge thread has no use for local scoped memories. The reason is, that local scoped memory is an attribute and wedge thread would have to have reference to the class, which owns that memory. Therefore the wedge thread flag is hidden in the right part of Figure 4.3.

```
startWedgeThread ( myMemory );

stopWedgeThread ( myMemory );
```

Figure 4.9: Appearance of starting and stopping wedge thread

4.4.2 Communication between Threads

Every complex RT system needs some way how the threads can communicate among each other. Standard solution for this issue is a channel, on its one side sender pushes some messages and on the other side receiver pulls these messages. This pattern is in normal programming world quite easy, but in real-time system there are some troubles needed to be solved.

First complication is a restriction coming from utilizing memories. We must ensure that channel is functional among any threads, regardless in which memory they run. The scoped memory cannot be used as a channel's repository for messages. If there is a thread running in immortal memory, it could not use reference to a message from channel. Because that message would be saved in scoped memory and reference would violate restriction of references among memories, as it was described in Section 1.4. Therefore messages must be saved in immortal memory.

This decision, however, has other limitations. We cannot allocate each new messages as a new object in this repository, because objects in immortal memory are not deallocated during run of program. We create a pool of objects of expected type. When sender pushes a new message into channel, we will create a copy of this message and save it in this pool. When receiver pulls this message from channel, we will deliver a reference to the object stored in our pool. This solution contains a few properties, which deserves more detailed contemplation.

Why copy of message is needed and not only a reference as usually, when we program in Java? Again because of the type of restriction of memory. The repository is located in immortal memory and sender could operate in the scoped memory. Thereby object must be copied, this is a little bit problematic in Java, the method `clone` must be used. This means that the message's class has to implement an interface `clonable`. But user can choose whatever he wants as a message's class. The exact solution, how to force the user to implement the clone method and not to restrict him too much, would be analyzed in chapter about DSL's implementation.

As we have chosen the repository for messages as limited object pool, the user has only limited time to work with message's reference. It is due to the fact, that this delivered message will be overwritten by a new one in the future. But messages have to implement the clone method, as we wrote in the previous paragraph, so the user can simply copy message in the moment of receiving it.

The last decision is what should the system do, when sender is sending a message to full pool or receiver is trying to pull a message from an empty pool. We decided to choose an asynchronous way. The possibility of blocking thread because of attempt to send message to full channel we find dangerous in real-time systems. Additionally the user has a possibility to simulate synchrony behavior by periodic thread.

Besides of this analyzed solution RTEJ offers two other implementations defined by RTSJ - `WaitFreeWriteQueue` and `WaitFreeReadQueue`. `WaitFreeWriteQueue` is non-blocking for consumers and is intended for single-writer and multiple-reader communication. `WaitFreeReadQueue` is designed in contrary a way of communication. For higher level of user-friendly handling, RTEJ provides these different implementations as enumeration selection for one common root concept, which defines the new communication channel. Additionally to this

enum, the concept requires a type of messages and the size of channel. Detail could be seen in Figure 4.10

```
Channel myChannel
  implementation type: OneProducentMoreConsuments
  type of messages: MessageObject
  number of messages: 50
```

Figure 4.10: Appearance of channel's declaration

Concepts for access to instantiated channel are independent of chosen implementation. There are three concepts:

1. `pushIntoChannel` - statement, which requires object and channel, to which is that object pushed;
2. `popFromChannel` - expression, which requires channel and returns object from channel or null, when is channel empty;
3. `isChannelEmpty` - expression, which requires channel and returns boolean value, if the channel is empty.

Sample usage is in Figure 4.11.

```
pushIntoChannel ( myChannel , new MessageObject()) ;

MessageObject message = popFromChannel ( myChannel);

boolean isEmpty = isChannelEmpty ( myChannel );
```

Figure 4.11: Appearance of concepts for handling with channel

4.4.3 Object Pool

Objects allocated in immortal memory have many advantages. They are accessible from all memory types, there is no need to use wedge thread. But there is the problem with immortality as was mentioned several-times. So it would be handy to have a system, which provides some pool of objects of given types. The user can get new object from this pool, work with it and after finishing working with the object he can call a method similar to `free`.

Like in several other problems described in previous sections three concepts should be specified - definition of object pool, get object and free object. The root concept fits great for the first one like for threads or memory. The concept requires the user to define the type of objects and the size of pool. Besides of these essential definitions the concept provides a flag, whether pool could be expanded during run of system. The second concept `getFromObjectPool` is straightforward - it is an expression with the name of object pool as the only parameter. Sample of usage of both concepts are presented in Figure 4.12.

Free object concept is a little bit complicated, because the system needs to know which element should be freed from which object pool. There are three ways, how to deliver these data to object pool.

1. concept requires both reference to element and object pool;

2. concept requires only reference to element, element holds reference to object pool in its data field;

3. concept requires only reference to element, reference to object pool is found by data-flow analysis.

The first solution is the easiest way to implement but the most annoying for the user, because he has to know, which element belongs to which object pool.

The second solution is more user friendly but element needs to know from which object pool it comes from. In normal programming it would be necessary to ask user to implement an interface. But RTEJ is a DSL and within the generation of code a wrapper class can be created, which implements the demanded methods. This wrapper class must extend the origin user's class. The user's comfort is reason for this, because as derived class the wrapper can access to all protected members and type-checking works. Of course, the origin class must allow extending, it means it must not be `final`.

The third solution is most complicated for implementation. For the user it is the best way, because there is neither knowledge of element's origin nor restriction for using only final classes needed. Implementing data-flow especially in this case would be too complicated. Because it is necessary not only to generate some warnings but to generate code in dependence on results of this analysis. Therefore we decided to choose the second solution. Example is shown in Figure 4.12.

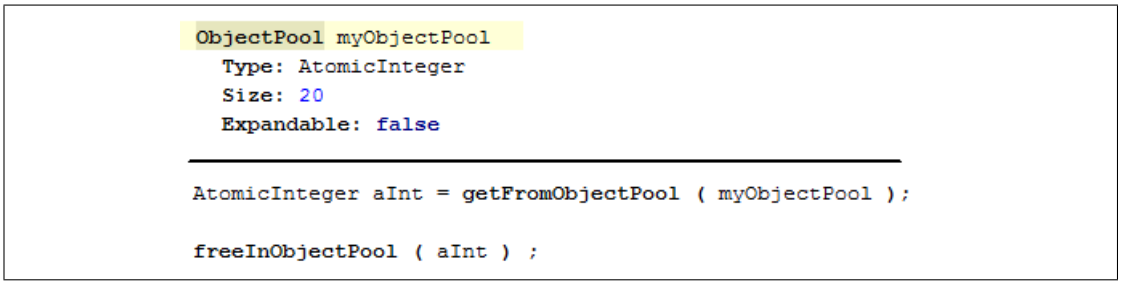

Figure 4.12: Appearance of concepts for handling with channel

4.5 Data Flow

One of the thesis goal is, that RTEJ should provide a controller of memory references to the user. The controller should validate correctness of object referencing among different memory area types during editing. MPS offers a possibility to write a data flow analyzer, which perfectly fit to this purpose.

The new memory assigning analyzer should maintain records about references of variables. These records can be changed by several ways

1. allocating a new object in specified memory area by RTEJ statements `AllocateVariable` or `AllocateArray`;

2. allocating a new object by Java statement `new`;

3. assigning reference to already existing object.

The first manner is straightforward, memory area of a new object is defined in used statement. The second one must be analyzed more deeply, because it

depends on the actual source for object allocation. The default source is heap. It can be modified by using RTEJ statement `EnterMemory`, which executes received code in the defined memory area. Each allocation in this code behaves like `AllocateVariable` or `AllocateArray`. Besides of `EnterMemory`, the source for allocation can be defined by the default memory area for realtime thread. This default area replaces the default heap source for allocating in the whole thread analogically to `EnterMemory`. To find default memory area of thread is very hard, because the `Runnable` object injected into `RealtimeThread` constructor does not know memory area of this thread. Moreover, instances of one `Runnable` class can be used as logic for realtime threads with different memory areas. Therefore RTEJ offers an opportunity to define default memory area of class in the similar way to annotations as it is shown in Figure 4.13.

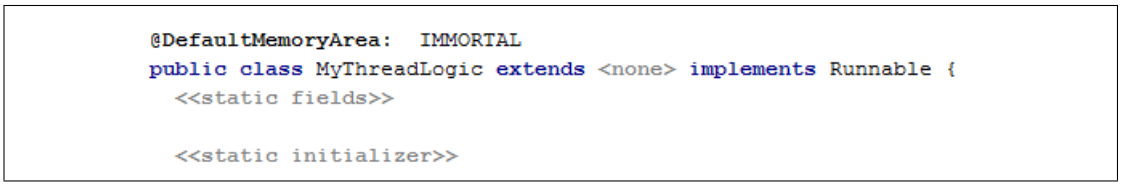

Figure 4.13: Appearance of default memory area of class.

The MPS analyzer should ensure the third type of assigning. Analyzer holds memory areas, in which the objects are allocated, and checks correctness of every assigning. In the case of assigning restrictions violation, the solution editor should mark the assignment with a warning. The main reason for warnings instead of errors is the above described problem of two instances of one object used in more memory areas. These solution ensures, that code can be always built.

5. Implementation of DSL

Implementation is split into the models of MPS. Basic rules of RTEJ language are described in the models Structure and Editor. Concepts appearance is defined in these models. Moreover, their basic info is defined here like name, ancestor and from what they are consisting.

Models Behavior, Type system and Constraints are covering behavior of language. Constraints like permitted parents of concepts in AST or types of returning values for type checking in solution are defined here.

Models Intention and Data flow are used for implementing better comfort for the user of RTEJ. There are several AST transformations in Intention model, e.g., conversion from one type of real-time thread to another. Data flow serves for static analysis.

The last used model is Generator. There are placed all constructs needed for code generation. As the main point of generated code is the class `MainDefinition`. It serves like central node, there are methods which are used by generated code for other concepts. Furthermore, there is Java method `main`.

So called virtual packages are used in every model, they serve for better structuring of the overall solution. They behave like normal Java packages, but only for source viewer, the generated code is in one directory.

Documentation of RTEJ directly in its code is not as extensive as we would like. The aspects of models with complex layout have not many possibilities for documentation. For example there is no opportunity in mapping layout how to add some comments for describing individual rules. Fortunately, there exists language `jetbrains.mps.baseLanguage.javadoc`, which implements the ability of adding classic JavaDoc to normal code. With help of this language, classes and methods in the Generator and the Behavior model were documented. In other models, like Constraint or Type system, at least the line comments were inserted. Descriptions of individual concepts have been added by property `shortDescription` in the Structure model. These descriptions are shown to the user in contextual menu in the solution editor.

List of all concepts is in Appendix D.

5.1 Structure and Editor

5.1.1 Non-root Concepts

Non-root concepts used directly in solution code should be as similar to regular Java statements as possible. So RTEJ are based on existed concepts of the base language like statements, expressions, field declarations etc. As an example the concept `AllocateVariable` is analyzed in further paragraphs. This concept serves for allocating a new object in specific memory area. The parent concept is `Expression`. Thanks that, this concept can be used wherever is expected an expression. In Chapter 4.2.2 was analyzed the necessity of restriction of the concept's usage only to the declaration concepts. This restriction is done in Constraint model, which is described in Section 5.2. The Structure aspect looks like Figure 5.1.

```
concept AllocateVariable extends    Expression
                         implements <none>

  instance can be root: false

  properties:
  << ... >>

  children:
  MemoryAreaReference | memory    | 1 | specializes: | <none>
  Type                | allocater | 1 | specializes: | <none>

  references:
  << ... >>

  concept properties:
  alias            = allocateVariableInMemory
  shortDescription = rtej: Allocate object in memory
```

Figure 5.1: Structure aspect of concept AllocateVariable

The Figure depicts the name of concept, extended parent concept, children and alias. Children are sub-nodes of this concept. Numbers in their rows represent the cardinality of using. An alias is a concept's name for solution code, both for auto-completing code and help in the context menu. Short description is shown in context menu in the solution editor as small info about concept. In the Editor aspect concept's looks for the solution editor is defined, it has been done as similar to normal method's call. Result can be seen in Figure 5.2.

```
editor for concept AllocateVariable
  node cell layout:
    [> allocateVariable ( % memory % , % allocater % ) <]

  inspected cell layout:
    <choose cell model>
```

Figure 5.2: Editor aspect of concept AllocateVariable

There is a special symbols for defining layout - "[>", which represents horizontal collection. Further, there are situated children, wrapped by %. There are many other chars like parenthesis, equal sign and string "allocateVariable". They are all constants, their job is only to create more understandable looks. The final layout in solution code, which user will see, is shown in Figure 5.3.

```
AtomicInteger i = allocateVariableInMemory
                  allocateArrayInMemory        rtej: Allocate array in memory
                  allocateVariableInMemory     rtej: Allocate object in memory

AtomicInteger i = allocateVariable ( <no memory> , <type> );

AtomicInteger i = allocateVariable ( immortalMemory , AtomicInteger );
i.addAndGet(5);
System.out.print(i.get());
```

Figure 5.3: Concept AllocateVariable used in solution code.

In the top part of the figure, the help in context menu is shown. In the middle part of figure concept is presented, as it looks like immediately after its inserting into code. The red underlined fields must be filled in by the user. The necessity of filling these fields was set by minimal value of the children' cardinality in the aspect Structure shown in Figure 5.1. The bottom part of figure introduces final looks in code with correctly filled fields. The type and name of variable is independent to concept `allocateVariable`, they are set as in normal local variable declaration, concept `allocateVariable` is only as initializer of this declaration.

5.1.2 Root Concepts

Root concepts serve for definition of various elements of real-time systems. The structure aspect is analogical to concepts from previous Section 5.1.1 with one change - root concepts are extending `BaseConcept`. A layout constructed in editor concept is designed as structured list of definition. Concept `PeriodicThread` is shown as a representative of root-concepts in Figure 5.4.

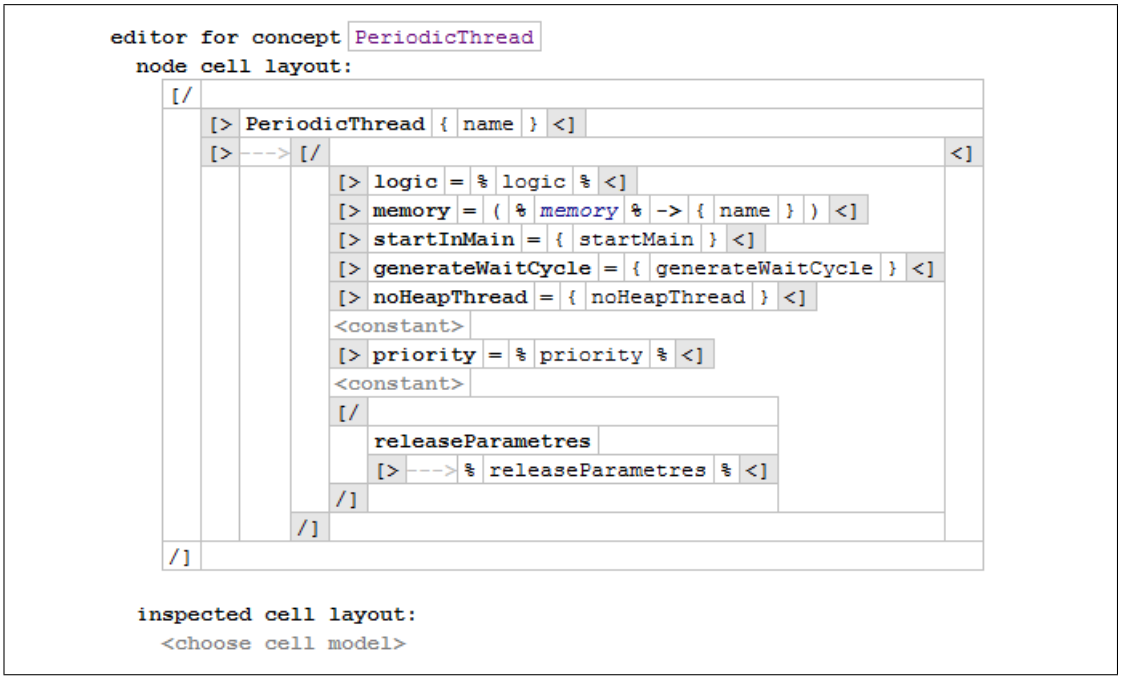

Figure 5.4: Editor aspect of concept PeriodicThread

More symbols for defining layout are used in Editor aspect of this concept, "[/" for vertical collection and "–>" for indentation. In this concept grey fields "<constant>" are used as blank rows, but normally they are filled with some text to display it. Additional, properties wrapped by braces are placed there. Reference to `MemoryArea` concept is required as memory, this reference is done by name. The rest of elements are the same as in previous Section 5.1.1.

Already presented Figure 4.1 shows using `PeriodicThred` in solution code. Left part displays situation after creating this concept. There are some red fields as in previous section, which must be filled. The grey ones are optional. In the right part there are filled all required fields and it can noticed, that user can use not only primitive types like integer but even objects from the rest of application.

In this case the user filled the name of existed memory `myScopedMemory` area in field "memory" and a new instance if class `ThreadLogic` in field "logic".

5.2 Constraint

The constraint model is used for various restrictions for instances of concepts, e.g. for which concepts this instance can be in relation as a child, parent or ancestor. Moreover, the restrictions for reference's scope or value of properties can be defined in this model. Both child limitations and reference restrictions are used in RTEJ.

Concept `AllocateVariable` can be used only in `LocalVariableDeclaration`, `FieldDeclaration` and `StaticFieldDeclaration`. It means that it can be instantiated as a child only for these concepts. Implementation is rather simple, there is only a condition checking the type of parent node. In dependence on its result the condition returns true or false value. Code is shown in Figure 5.5.

```
concepts constraints AllocateVariable {
  can be child
    (operationContext, scope, parentNode, link, childConcept)->boolean {
      // only in declaration of new variable or attribute
      if (parentNode.isInstanceOf(LocalVariableDeclaration) ||
          parentNode.isInstanceOf(FieldDeclaration) ||
          parentNode.isInstanceOf(StaticFieldDeclaration)) {
        return true;
      } else {
        return false;
      }
    }
```

Figure 5.5: Constraint aspect of concept AllocateVariable

Restrictions of possible references are shown in several concepts. Concept `WedgeThreadStart` requires as a reference the concept `ScopedMemory`, which has set flag for wedge thread as true. Therefore restriction for this reference returns a list of all instances of `ScopedMemory` in application filtered by condition for mentioned flag.

```
concepts constraints WedgeThreadStart {
  link {memory}
    referent set handler:<none>
    search scope:
      (model, scope, referenceNode, linkTarget, enclosingNode, operationContext)
        return model.nodes(ScopedMemory).where({~it => it.wedgeThread; });
      }
    validator:
      <default>
    presentation :
      <no presentation>
  ;
```

Figure 5.6: Constraint aspect of concept WedgeThreadStart

The more complex sample of restriction is shown in Apendix C, which contains a filter for applicable memory references used in other concepts like `EnterMemory`

or `AllocateVariable`. In that case, various flags must be checked and the problem with local scope solved.

5.3 Behavior

The behavior aspect serves for implementing methods for specific concepts. These methods behave like in normal OOP[1] language. They can be used in other models, especially in the generator. In RTEJ behavior methods are used mainly for for getting unique names of instances and collecting of instances, which fulfill certain conditions. Figure 5.7 depicts method for getting all instances of concept `ScopedMemory`, which is declared as a root-concept. When the concept is root, it has no ancestors. Variable `genContext` is context of generator.

```
public nlist<ScopedMemory> getRootScopedMemories(gencontext genContext) {
  nlist<ScopedMemory> allMemories =
      genContext.originalModel.nodes(ScopedMemory);
  nlist<ScopedMemory> result = new nlist<ScopedMemory>;

  foreach specificMemory in allMemories {
    // has no ancestors => it is root concept
    if (specificMemory.ancestors.size == 0) {
      result.add(specificMemory);
    }
  }

  return result;
}
```

Figure 5.7: Behavior method for getting root scoped memories.

5.4 Type System

The aspect type-system serves mainly to define types of concepts. Thanks to that, concepts are correctly type-checked in solution code. There is quite a complex system for sub-typing in MPS. It is the whole tree of sub-type relations, which is parallel to a classic tree of inheritance relations. Sub-type relations are based on inheritance relation, but can be modified. For example concept `IntegerConstant` does not have to be child of `DoubleConstant`, but the user can define `IntegerConstant` as a sub-type of `DoubleConstant` and then the solution code will accept IntegerConstant, when it requires `DoubleConstant`. There are several operators for defining these sub-type relations in MPS like ":<<=:" for strong sub-typing or ":==:" for type equation etc.

In Figure 5.8 it is created restriction, that field logic in definition of `Realtime-Thread` concept must implement interface `Runnable`. To be precise there has been created a control mechanism, which checks, if field `logic` of concept `Realtime-Thread` is a subtype of interface `Runnable`.

[1] Object Oriented Programming

```
rule typeof_RealtimeThread {
  applicable for concept = RealtimeThread as realtimeThread
  overrides false

  do {
    check(typeof(realtimeThread.logic) :<<=: <Runnable>);
  }
}
```

Figure 5.8: Restriction to logic of RealtimeThread.

Figure 5.9 is a classical example of defining type of concepts. The concept `interThreadChannelPop` returns a message from channel between threads. The type of this concept should be the same as the type of this message. The channel was defined sooner by the user and the type of message has been set, so RTEJ knows that type and this type-system equation finishes link between them.

```
rule typeof_InterThreadChannelPop {
  applicable for concept = InterThreadChannelPop as obj
  overrides false

  do {
    typeof(obj) :==: obj.channel.messageType.getUnboxedType();
  }
}
```

Figure 5.9: Setting type of concept interThreadChannelPush.

Concepts' types can be changed during user's editing. The type of concept `rawMemoryGet` depends on value of parameter `varType`. Determination of type is done after each change of any field of this concept.

Type system model is used for various checks besides of type checking described in previous paragraphs. There are implemented checks for unique name of concepts like threads, memories or object pools. Another restriction is used to guarantee that concept `MainDefinition` is instantiated only once. Type system model is extra suitable for these checks, because it can generate its own warnings and errors like normal errors shown during edition in IDE.

5.5 Generator

There is only one generator with one mapping configuration file in RTEJ. All transformation rules are defined in this file. Majority of rules described in Section 2.3.3 are used.

5.5.1 Non-root Concepts

Reduction rules are used for most of concepts especially for all in-code concepts. Most of them are simple node to node transformations, where each instance of a new RTEJ concept is transformed to concept from the Java language without any condition. Realization of these transformations is based on a template declaration with specific data from concept's instance included by macros. Reduction rules must observe a rule, that the result of transformation must have the same type

as the source node. That means, that concept extending expression must convert into other descendant of expression. But many single-statement concepts in RTEJ need to be transformed into a list of statements, these generated boilerplate code is often large. This problem is solved by calling a method, which executes all needed operations and returns the result. These helper-methods are implemented as static members of class `MainDefinition`, which is the main node of the whole generated application and is described in Section 5.5.3.

Example of the reduction rules is presented on concept `EnterMemory` in Figure 5.10. In the top of the figure is shown the mapping configuration and in the bottom the template declaration. As was written in previous paragraph, the logic itself is executed by static member of `MainDefinition`. Moreover, `SWITCH` macro is used for fill in the correct memory area. There are three types of memory area in code generated by RTEJ's generator:

- immortal memory - referenced as in RTSJ by `ImmortalMemory.instance()`;
- global scoped memory - saved as private attribute of `MainDefinition`;
- local scoped memory - saved as protected attribute of a specific class.

For each type a different code must be generated. Because memory areas are referenced from several places, the `SWITCH` macro is defined. Analogical technique is used for declaration of memory area, release parameters, logic of thread's logic, too.

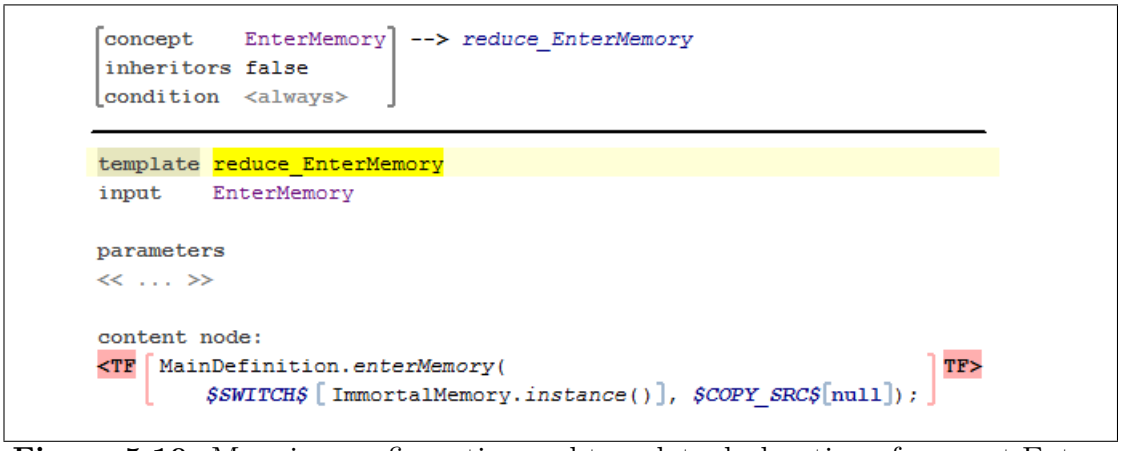

Figure 5.10: Mapping configuration and template declaration of concept EnterMemory.

Generating of concept `AllocateMemory` is the most complex generation of in-code concept. The local and global scoped memory must be differentiated. Global scoped memories are saved in `MainDefiniton`, attributes for saving them are declared during generation that class and allocating itself is simply done by calling an appropriate generated method. But local scoped memories are saved as attributes of classes, where the concepts `AllocateMemory` are placed. The RTEJ does not control generating process of these classes, thereby a weaving rule must be used. The weaving rule serves for adding a new child to already generated node. Additionally to the reduction rules, the weaving rules need a context. The context is the existed node, in which is the new child added. The upper part of Figure 5.11 shows declaration of weaving rule, the context is ancestor of generated

`AllocateMemory`, which has type as `ClassConcept`. In the bottom of the figure is the important part of template declaration.

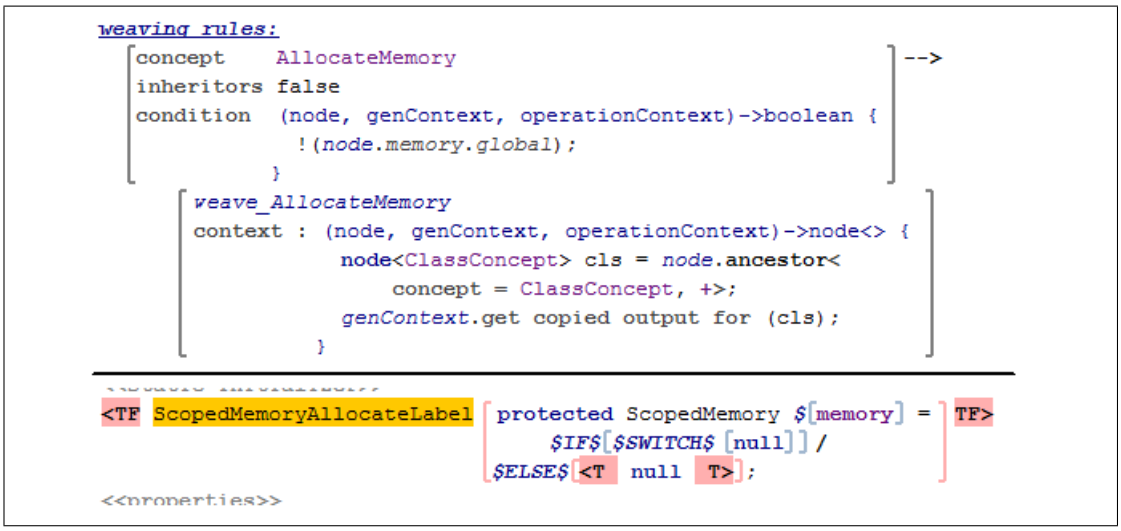

Figure 5.11: Weaving rules on concept AllocateMemory.

When the reference to the inserted node is used in the rest of application, it must be correctly linked. That is done by so called *mapping labels*. The mapping label serves as a map holding links among original concept instances and generated ones. The mapping label must be firstly declared as it is shown in the top of Figure 5.12. There are defined name of label, source concept and result concept. Then during generation the new concept, the template fragment is linked to specific label. That connection is displayed as an orange box in Figure 5.11. The last part of process is defining a reference macro with usage of the label to correct node as it is shown in lower part of Figure 5.12.

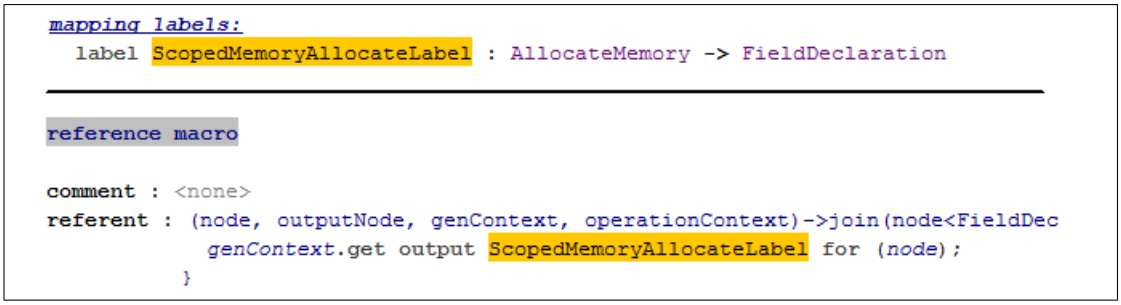

Figure 5.12: Usage of mapping label.

5.5.2 Root Concepts

The definition of a new structure as `RealtimeThread` and `ScopedMemory` are generated within other concepts, hence there are defined as abandon roots in mapping configuration. The concept `MainDefinition` is main root node of RTEJ, which takes care about another root concepts, thereby it is implemented as root mapping rule. In this group of rules there are special techniques for pattern rules, which will be discussed in Sections 5.5.5 and 5.5.6.

Besides these main groups of rules, the conditional root rules have been used for new classes and interfaces required by other concepts. Sample of usage is a

wrapper for `waitForNextPeiod` cycle. This class requires an instance of class implementing `Runnable`. This class has only one method - `run`. It contains a do-while cycle with calling method `run` on that received object. This class is neither generation nor transformation for any concept of RTEJ, but must be generated anyway. This generation is done by conditional root rules, which create a new root node. Another example of using this kind of rules is wedge thread, which is described in Section 5.5.4.

The last part of used configuration mapping are pre- and post- processing scripts, we have created one pre-processing script for initialization of objects needed for pattern concepts.

5.5.3 Main Definition

RTEJ needs a repository which takes care about all objects generated from the root concepts. Moreover, there is needed the main static class which should be launched at the start of application. Both these roles are ensured by generated class `MainDefinition`. This class is generated from concept of the same name by root mapping rule. The generation is based on template declaration, which is full of various macros.

The class is implemented as a standard main Java class. That means there is a static method `main`, which is launched by operation system. This static method allocates instances of RTSJ classes `RealtimeThread`, `ScopedMemory`, etc. with parameters defined by the user. Before and after this allocation are launched, the logic referred by the user in concept `MainDefinition` is executed. At the end of method `main` default mode is set and real-time threads are started, which are defined as starting threads by user.

Objects defined in method `main` must be saved in the application. As a repository for these objects could be utilized Java `HashMap`. But `HashMap` is not the best choice for real-time systems. It has nontrivial overhead and there is a possibility of memory leaking, because `MainDefinition` runs in immortal memory to ensure access from all types of memory areas. Therefore each instance of mentioned classes is saved as private attribute of the `MainDefinition`. Furthermore, there are generated methods for handling with these attributes. Names of these attributes and methods are created from prefix and name of specific instance of concept.

Methods described above are used in the rest of application's code. Their calling is generated by other template declaration, as was shown in Figure 5.10. We have decided to implement these methods as static. The reason is easier access from each position of code and elimination of referencing to the main object across the whole application. The calling of correct method is provided by a reference macro. Because the methods which are using attributes are static, the attributes themselves are static, too.

Besides of methods for handling with attributes, which are mainly simple getters and setters in several cases with try/catch block, there exists a method with generated larger code. This method is used for running mode and contains launching and stopping threads, changing their priority or even release parameters. Last unmentioned methods in `MainDefinition` are dedicated for handling RTSJ exceptions. These methods serve for catching RTSJ exceptions and trans-

lating them to Java exceptions or printing some readable text regarding errors from bad usage of RTEJ.

5.5.4 Wedge Thread

Wedge thread is implemented as a wrapper class. It extends class `Realtime-Thread`. During initialization the thread is created and stored a reference to the specific scoped memory. This class is generated by the conditional root rule, which creates this class as a new root node without necessity of instancing of any concept by user of RTEJ. During generation, instances of this class for each scoped memory with flag `wedgeThread` declared as true are created. Concept `WedgeThreadStart` causes the launching of thread. All job of the thread is to enter the memory and wait. Concept `WedgeThreadStop` causes an interrupt of thread.

5.5.5 Object Pool

Implementation of object pool is the most complex of all concepts in RTEJ. As was analyzed in Section 4.4.3, a repository and a wrapper for elements are needed . The repository is stored in `MainDefinition` as private attribute as other special objects like `RealtimeThread`. Creating of this repository's object and allocating is done during the static method `main`. Besides that there are methods for working with object pools - `getObjectPool` and `freeInObjetPool`. They are launched by other concepts.

The wrapper is a generated class, which implements interface `ObjectPool-Element`. The interface provides methods for getting and setting object pool of that element. The reference is necessary for marking that element in correct pool as free to next allocation.

The described interface is implemented by class `ObjectPoolElementImpl`. That class implements methods for utilizing with reference to object pool and extending class of user's element. Thus acts as wrapper and the user can use it as an origin element. Extending itself is easy, RTEJ substitutes the name of element's class for each user's class stored in object pools. Of course generated class must be named uniquely, in RTEJ there are used a static prefix and complete name of extended class.

It is possible, that user uses one element's type in more pools. In this case it is nonsense to generate `ObjectPoolElementImpl` for that class twice. Moreover, it would end with an error, because names of these generated classes would be equal. Thereby RTEJ creates a list of already created classes and check it during generation. This list is initialized in the pre-processing script. It is stored as a generation context session object, detail see in Figure 5.13.

```
mapping script script

script kind    : pre-process input model
modifies model : false

(model, genContext, operationContext)->void {
  genContext.session object [ "ObjectPoolElement" ] = new ArrayList<String>();
```

Figure 5.13: Declaration of session object

Generation is performed by root mapping rule with a condition, which ensures that each class used in object pool is generated just once. Detail of that rule is in Figure 5.14.

```
[concept         ObjectPool                                                    ]-->
[inheritors      false                                                         ]
[condition       (node, genContext, operationContext)->boolean {               ]
                  List list = ((ArrayList<String>)
                       genContext.session object [ "ObjectPool" ]);
                  String className = node.type.getClassExpression().
                      getDetailedPresentation();
                  if (list.contains(className)) {
                    return false;
                  } else {
                    list.add(className);
                    return true;
                  }
                 }
[keep input root default                                                        ]
    ObjectPoolImpl
```

Figure 5.14: Generation of classes for object pool's elements

Object pool's repository for elements is designed similarly. There is an interface `ObjectPoolI` with methods for getting and freeing objects. That interface is implemented by class `ObjectPoolImpl`. Implementation solves repository as two instances of `ArrayList`, one for elements themselves and the other as bitmap for marking which elements are free. Arrays would not be enough, because the user has a chance for dynamical enlarging of the pool.

The class `ObjectPoolImpl` is again only a template for each user's class used in object pools. Resulting generated classes allocate instances of the specific user's class as new elements. Creation of these generated classes proceeds in the same way as generation of elements' classes.

Method `getObjectPool` returns object pool with received name stored in `MainDefinition`, concept `ObjectPoolGet` uses that method and gets first free element from received object pool. Generated code typecasts returned element from general type `object` defined in interface to specific user's type. Possible warnings of bad type are removed by this action. That is correct, because generator knows the proper type of used object pool. The method is generated for each object pool, hence concept `ObjectPoolGet` calls method directly by name and there is no looking in any hashmap or similar.

Method `freeInObjectPool` calls on received element the method `getObjectPool` and on the object pool, which owns that element, calls method free. Described method is used by concept `ObjectPoolFree`, which simply calls method with reference to element to be released.

Only few things left. Methods in `MainDefinition` check, if received element is really coming from object pool. And both used interfaces must be generated, too. It is done by conditional root rules.

5.5.6 Communication Channel

The communication among threads is implemented as an object pool. Objects providing the communication itself are saved as private attributes in `MainDefinition`. Initialization takes place again during the main initialization in method `main`. For handling with the specific channel there are generated methods for push message, pop message and for find out if the channel is empty. These methods solve problem of different implementations, which are chosen for specific channel. Method's code is generated already for the specific channel's implementation. In following paragraphs is described implementation of own RTEJ channel implementation. For two other ones are used only existed interfaces defined by RTSJ.

In contrast with the object pool, elements are purely created by the user. There is not any generation, but RTEJ requires, that user's elements must implement interface `CommunicationChannelElement`. This interface is parametrized by type of object and define method `clone`, which demand an object as parameter. The implementation of this method should clone actual instance to received object.

The channel's repository is defined by interface `CommunicationChannel`, which define methods `push`, `pop` and `isEmpty`. These methods work with objects implementing interface `CommunicationChannelElement`. The pool for messages is created as fixed array with two pointers to position of last inserted and last read elements. Method `isEmpty` simple return if the channel is actually empty.

Mentioned interface for elements must be accessible both for the user in his solutions and during generation. The interface created in the Generator model can be used during generation but cannot be referenced in the solution editor even after including the Generator model. The reason is, that after code generation the generated interface has the same package as the rest of code, but code references interface from generator model - from generator's package. Thereby this technique is functional only during editing and building fails on these broken references. Functional solution consists of creating a java stub of this interface and including it to the solution editor. But it is not much user-friendly, because the user has to include this stub in his solution, and it must be distributed with RTEJ.

Hence, we have decided to solve problem in another way. The new concept has been created, with the same name as interface. This concept is descendant of interface and user must instance it in his solution. The concept is populated on itself with correct data and structures after creating. The user can see and use the concept as a normal interface, he only cannot edit it.

Similar as in object pool, the class `CommunicationChannelImpl` is only a template, from which generator creates classes for each type of user's channels. There is a greater possibility than in object pool, that user will define more channels with the same type. Therefore the mechanism with session object is used again.

The sequence of methods' calling during usage is following. Code generated from concept `CommunicationChannelPush` calls appropriate generated method `pushToChannel` and hands over the message to sent to this channel. This message is cloned to place in pool, for copying is used the method of message. After that is message safely stored in immortal memory. Receiver uses code generated

from concept `CommunicationChannelPop`, which calls specific generated method `popFromChannel`, which returns a reference from specific channel to first unprocessed message. The generation of code for receiving message ends here, but user should clone that message as soon as possible. The reason is, that the channel will overwrite this message by a new one. Nevertheless, this cloning is possible with the same method, which was used by channel, so it is not a problem.

5.6 Intention

There are several intentions in RTEJ for increasing level of user's comfort. Both "Surround with" possibility and transformation from one concept to another are used.

When the user marks statements he can surround them with concept `EnterMemory`. The contextual menu is shown in upper part of Figure 5.15, in lower part is shown code after transformation.

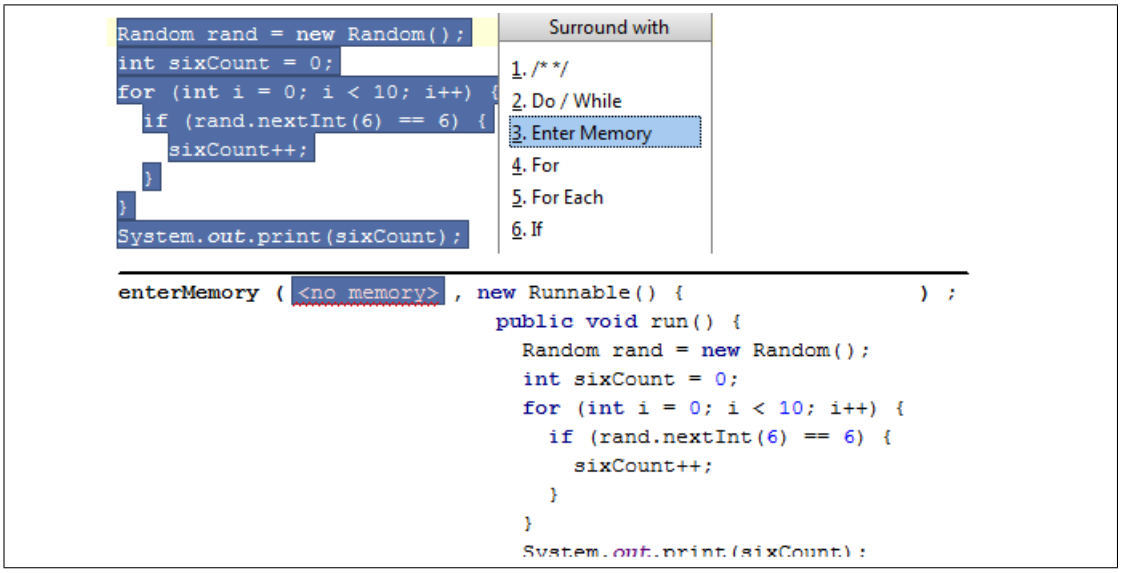

Figure 5.15: Intention for surround With concept EnterMemory.

The second used type of intention is transformation from one concept to another. For better change of thread's type there are transformations among types in RTEJ. These transformations create a new root node as an instance of required thread's type in the model of existing thread. After that all possible definitions are copied from the old thread and at the end is old one deleted. In total there are six threads swapping.

Besides swapping there are other helping intentions, e.g., creating new anonymous `Runnable` interface into `EnterMemory`, so user does not have to write it himself. The possibility of intentions is shown by yellow bulb, as it is displayed in Figure 5.16.

Figure 5.16: Intention for adding Runnable to EnterMemory.

Figure 5.17 describes implementation of intention of surrounding with concept `EnterMemory`. The execution consists of creating a new resulting concept. Further, the selected statements are added as the body of new anonymous `Runnable` interface. This interface is created in method `getNewRunnable` and is not shown, because it consists of quite large uninteresting piece of code of defining several sub-nodes of this interface. The most important thing is to add this new statement into AST with correct connections. That is done by `add next-sibling` statement. Besides described implementation each intention needs a name, for which concept is used and its description. This description is shown to user in contextual menu.

```
execute(editorContext, node)->void {
  // create new enterMemory statement
  node<EnterMemory> enterMemoryStatement
      = new initialized node<EnterMemory>();
  nlist<> selectedNodes = editorContext.getSelectedNodes();
  // join new statement to AST
  node.add next-sibling(enterMemoryStatement);

  // add all selected nodes to new body of creating statement
  node<StatementList> body = new node<StatementList>();
  foreach selectedNode in selectedNodes {
    body.statement.add(selectedNode.ancestor<concept = Statement, +>);
  }
  // add new runnable to enterMemory statement
  enterMemoryStatement.runnable = enterMemoryStatement.getNewRunnable(body);
  // focus editor to memory attribute of new statement
  editorContext.select(enterMemoryStatement.memory);
}
```

Figure 5.17: Implementation of intention for surrounding with EnterMemory.

5.7 Data Flow

5.7.1 Analyzer for Memory Areas

The Data flow aspect is used primarily for validation of allocation in memory areas in RTEJ. `MemoryAreaState` is an enumeration for defining source memory area of node. It has five possible values

- *NOT_INIT* - a node, which is not processed yet, or does not concern to memory areas;
- *HEAP* - a node allocated in heap;
- *SCOPED* - a node allocate in scoped memory;
- *IMMORTAL* - a node allocated in immortal memory;
- *RULE_VIOLATION* - a node, which violates restrictions of assignment.

Besides these states, `MemoryAreaState` contains a method for calculation of the result state, when two nodes are merged. That means, when parent's state is calculated from states of its children.

Initial states are defined by rules for data flow analyzer. These rules are defined for specific concepts and can insert newly defined instructions. In RTEJ, rules for variable declarations are defined. According to type of declaration the instruction is inserted. These instructions are corresponding to memory area states, as they were defined in previous paragraph. Algorithm for choosing instruction consists of comparing locations of reference and allocated object. Location of reference is found from default memory area for its class and possible usage of `EnterMemory`. Location of object is determined analogically with possibility of using `AllocateVariable` or `AllocateArray`. Instruction adequately to `RULE_VIOLATION` state is inserted, when the rules for assigning among memory will find a violation.

The `MemoryAreaAnalyzer` analyzer constructs data flow graph of instructions for a specific concept. The graph is represented as map of instructions and states `MemoryAreaState`. The graph is constructed by composing of instructions of each node and merging their states. That is possible, because every concept can have defined data flow representation. Data flow representation for `LocalVariableDeclaration` concept from the base language is shown in Figure 5.18. That script inserts code for initializer and assigns it to node, when the initializer exists. In the base language there are much more complex data flow representations, i.e., concept `ForStatement`.

```
data flow builder for LocalVariableDeclaration {
  (node)->void {
    nop
    if (node.initializer.isNotNull) {
      code for node.initializer
      write node = node.initializer
    }
  }
}
```

Figure 5.18: Data flow representation of concept LocalVariableDeclaration.

Warnings are generated by checking rule `check_MemoryAreaReferencing`. That rule is a part of the type system model and is applicable for every concept, which could violate memory reference restrictions. The rule constructs data flow graph for concept by calling `MemoryAreaAnalyzer`. The result of analyzer is tested for RTEJ instructions, which represents `MemoryAreaState`. When the `RULE_VIOLATION` is found, the warning is created for the appropriate statement.

5.7.2 Default Memory Area for Class

The user needs an option how to define the default memory area of a class. It could be done by the Java annotation. But MPS provides a special concept `NodeAttribute`. This concept allows to alter a concept from another language without modifying its structure. RTEJ cannot directly edit `ClassConcept`, which is an element of the base language, therefore `NodeAttribute` is a perfect choice. Newly created concept `ClassMemoryArea`, which extends mentioned concept, defines a role and attributed link in the structure model. The attributed link is the concept, which is altered, and the role is name for this new attribute of the altered concept. It is possible to access to the elements of this attribute node by

the role name. `ClassMemoryArea` has only one property, which is enumeration for memory area type. In Figure reffig:nodeAttributeUsage is shown an example of access to the value of `ClassMemoryArea`. This code is a part of analyzer rule for finding position of reference in declaration of variable.

```
// class annotation
node<ClassConcept> classAncestor =
    variableDeclaration.ancestor<concept = ClassConcept>;
if (classAncestor.isNotNull) {
  // class is annotated with type of memory
  if (classAncestor.@memoryScope.isNotNull) {
    // not in enter memory
    if (refPos.equals(MemoryAreaState.HEAP)) {
      if (classAncestor.@memoryScope.type.is(< IMMORTAL >)) {
        refPos = MemoryAreaState.IMMORTAL;
      } else if (classAncestor.@memoryScope.type.is(< SCOPED >)) {
        refPos = MemoryAreaState.SCOPED;
      }
    }
  }
}
```

Figure 5.19: Data flow representation of concept LocalVariableDeclaration.

The editor aspect of this concept is very simple. The only important thing is to insert a cell `attributed node`, which ensures including altered node. In RTEJ's case it is the `ClassConcept`. The inserting of `ClassMemoryArea` is done by the intention aspect for the `ClassConcept`, it is created as a toggle.

6. Case Study of the DSL's Usage

6.1 Introduction

The example for evaluation and testing of RTEJ is based on a sweet factory [5]. This sweet factory is an example published by IBM, which serves to introduce several techniques and recommendations to handle realtime systems with using RTSJ[1].

The sweet factory example follows the original design. But the new implementation uses RTEJ instead of RTSJ. That means, the RTEJ concepts are used whenever the real-time structure is needed. Moreover, presented real-time design patterns like object pool and communication channel are utilized. Thanks to that, it is possible to evaluate the power and user-friendly level of RTEJ by comparison with RTSJ.

6.2 Design

The basic idea is a production line in factory. There are created various sized jars with different types of sweet. These jars are weighed and when the weight of the jar is not in tolerance, the supervisor is alerted. Moreover, there should be an auditor, which saves information how many jars was produced.

The calculation of correct weight is not trivial, in addition jars can be produced very quickly, even each 10 ms. So the algorithm for detection whether the specific jar was reported due to bad weight should be placed in a dedicated thread. Have a dedicated thread for each jar is not a good solution, because allocation of new thread implies a significant overhead. Therefore the example uses a pool of threads. From this pool a free thread is taken for each jar. When the pool is empty, a new thread is created and inserted into the pool.

Both threads for serving each production lines and thread for calculation of proper weight of jars have critical deadlines, because jars should not be missed. Thereby they must not be interrupted by the garbage collector and they should be implemented as `NoHeapRealtimeThread`. In the contrast, the thread for logging individual jars and counting totals could be implemented as normal thread, because writing to the log can wait a few milliseconds without any harm.

The threads need to send information about jars. We must analyze, where this object should be stored. Heap cannot be used due to `NoHeapRealtimeThreads`, usage of scoped memory is impossible too, because it could break restriction of references. So these objects must be saved in immortal memory. To avoid memory leaks, the pattern object pool is used.

The whole design is shown in Figure 6.1 [2].

[1] The referenced article, in which sweet factory example was presented, is a part of series on theme writing and deploying real-time Java applications and brings a lot of interesting information.

[2] Figure 6.1 has been based on the image from original implementation of the sweet factory [5]

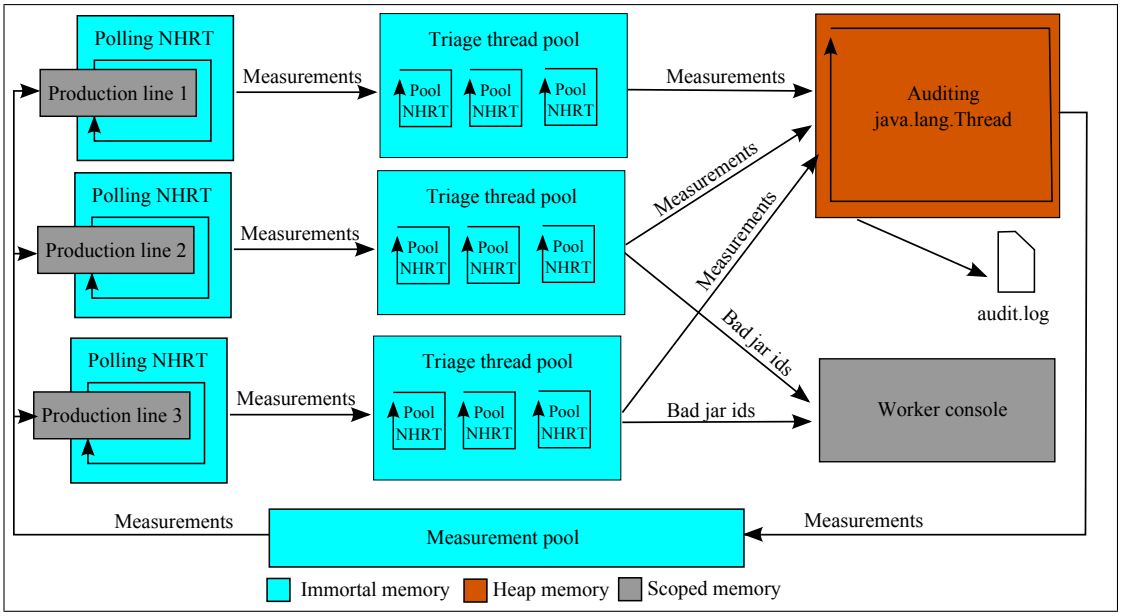

Figure 6.1: Design of the sweet factory

6.3 Implementation

The class `MonitoringSystemImpl` represents an entry point of the whole system. Methods, from the implemented interface `MonitoringSystem`, enable to start and stop the example. The instance of this class holds references to all objects in the rest of application - production lines, worker console, audit log etc. Its methods mainly allocate appropriate variables and run threads, analogically the stop method serves for stopping all work and prints a summary.

The class `ProductionLinePoller` is determined to maintain production lines. It gets all data about jars from physical production line, in this example simulated by `DummyProductionLine`. After receiving data about a jar, `ProductionLinePoller` hands data to its own `JarTriage` to process it.

`JarTriage` contains pool of thread for calculating the correct weight of specific jar. `JarTriage` has these calculating threads as nested class `TriageRunnable`. `JarTriage` chooses a free thread from the pool for received measurement data and forwards this data to thread. When no thread is free, `JarTriage` creates a new one. `TriageRunnable` calculates correct weight of jar according to its size type and sweet type. Thread compares the actual weight of jar to this correct value and when the difference is out of tolerance, relevant method on worker console is called.

In real world the worker console would be a physical device, which should display error messages to human operator and would provide some possibilities to react to this situation. In implemented example this console is created as class `DummyWorkerConsole`, which implements interface `WorkerConsole`. Methods defined in this interface only write error messages to error output and increment a counter of underfilled respectively overfilled jars.

`AuditLog` serves for saving or printing information about each jar. This job is not time critical, therefore it is a normal thread. For trouble-free communication with the triage threads, which have no access to heap, there is communication channel `WaitFreeWriteQueue`, which removes problem with priority inversions.

This class has method `log`, which saves received data object to channel. Moreover there is created nested class `AuditingRunnable`, which continuously reads messages from that channel and processes them.

Data about jars, which is being sent in the whole application, is represented by class `Measurement`, which is a typical data class. Instances of this class are saved in immortal memory, as was explained in previous Section 6.2. For that purpose there is the class `MeasurementManager`, which implements an object pool and provides method to get and free instances of `Measurement` class.

6.4 Usage of RTEJ in Example

RTEJ's concepts are used for basic real-time operations, like allocating real-time threads with `allocateThread`, allocating scoped memories with `allocateMemory` or allocating variables in a specific memory with concepts `allocateVariable` and `allocateArray`. Besides of these clear statements, which serve mostly for easier settings of parameters, there are used concepts for removing boilerplate code like `EnterMemory`. From pattern concepts the object pool and the communication channel are used. `MeasurementManager` is implemented as instance of object pool concept and one communication channel is used in class `AuditLog`.

The example utilizes only most important RTEJ constructs by neglecting the constructs which are not necessary for implementation. The implementation showed a problem regarding the local memories allocation. Impossibility of allocation the scoped memory for each instance of class have caused changes in RTEJ design. The flag local/global removes this limitations. Thanks to that the implementation of problematic parts in example was possible and easy.

The implementation process has begun by creating instances of singleton-like concepts `MainDefinition`, `CommunicationChannelElement` and `ImmortalMemory`. After that one communication channel and one object pool were implemented, as was described in previous section. The rest of implantation was identical with ordinary Java application.

7. Evaluation

7.1 Evaluation of RTEJ

Evaluation of RTEJ is based on two criteria:

- comparison to RTSJ;
  - which structures of RTSJ are covered and how;
  - additional abstractions;
  - complexity of learning RTEJ;
  - assembling and deploying created application;
- evaluation of implemented example;
  - efficiency of development;
  - clearness of final code;
  - analysis identified problems.

7.1.1 Comparison to RTSJ

RTEJ is actually a subset of RTSJ features with several additional conceptual extensions. Firstly lets review how many structures of RTSJ has been implemented in RTEJ. The majority of classes connected with main purpose of RTSJ, the memory areas and threads, are transformed into RTEJ's concepts. Classes, which are not implemented, provide advanced and complex settings like `Scheduler` for alternative scheduling policies or `RealtimeSystem` for setting maximum of concurrently locks or accessing to the garbage collector. As well possibility for working with time was restricted, the granularity to microseconds has been found as superfluous.

Besides of adopted structures from RTSJ, several advanced concepts have been added. Design patterns and system for modes belong among these concepts belong. In RTSJ there is only a pattern communication channel among threads, so these new patterns could really help user by saving his time and effort.

During designing RTEJ's concepts the emphasis was placed on clearness at the expense of variability. Therefore variability of some attributes have been restricted. For example RTEJ's concept for release parameters requires ordinal integer for deadline, cost, start time etc. While RTSJ requires `RelativeTime` object with both milliseconds and microseconds as parameters. This restriction of rarely used features has been used during designing RTEJ several times. Thanks that the concepts are well-arranged and the user can easily understand the whole created system or find typing error than it is in more complex RTSJ definition. The separation of definition of main structures like threads and memory areas to dedicated root concepts has the same purpose.

Additional to previous described changes there are quick AST transformations and consistency validations. These features should make developing more efficiency than in pure RTSJ.

Despite of all efforts, the creation of a fully functional data flow analyzer of memory area referencing has not been successful. Analyzer does not work correctly, when the already existed object is assigned to another variable. Nevertheless, RTEJ is creating warnings for declaration of a new object. Influence of `EnterMemory` and default class memory area are included, too.

Based on implementation of example the difficulty of learning of RTEJ's using is considered lower than learning of RTSJ. A problem may be handling Jetbrains MPS, editing tree structure instead of text is a little bit confusing at the beginning. However, with some practice the developing is as fast as in classic IDE. Installation of RTEJ is trivial, only copying two jars to correct place, detailed instructions are in Appendix B. Building of solution is simple, too, because MPS builds classes automatically after the code generation.

7.1.2 Experiences from Example

Implementation of the example shows, that RTEJ can used for developing real-time system. Clearness of used concepts is good. Non-trivial amount of time was saved because an object pool for measurements does not have to be created. In the original example it was implemented by a dedicated class `MeasurementManager`, but in RTEJ-based version the issue was easily solved bu ysing the RTEJ construct. Additional work with communication channel was easy, too.

However, problems related to objects with local effect appeared. The example has demanded local scoped memories. Therefore the original idea of memory areas had to be reanalyzed and this opportunity had to be added. Actually situation of memory areas is sufficient for implementing the example. Nevertheless, there is a possible situation, where the chosen solution would be too restrictive. But another changes from root-concept systems to classically solution with variable and types would totally ruin advantages of clearness and efficiency, which is one of the goals of this thesis. So actual solution is a good compromise.

Generated code has the same layout as it has in the solution editor. Therefore the user can further edit it and compile on its own. But we have used built files from MPS, which are compiled simultaneously with the code generation. That bytecode was executed on virtual machine Sun Java Real-Time System[1]. Resulted program worked correctly.

7.2 Evaluation of JetBrains MPS

To evaluate power of MPS as a DSL development platform there is necessary to analyze support for each step of developing process. Firstly it should be described how the MPS is efficient for creating of DSL. The next step is deployment. There is important to discuss, how the created DSL can be used in other IDE and how it is possible to build code using this DSL.

Each IDE provides support from its developers. Both for help with realization of its features and fixing bugs or future extending. Of course, it is not possible to cover all concepts of MPS by one project. Therefore there are described used

[1] http://java.sun.com/javase/technologies/realtime/index.jsp

and unused parts by RTEJ in this section. Version of MPS, which was evaluated and used in this thesis, is 2.0.6 (build 7767).

7.2.1 User Interface of MPS

MPS IDE can be split into three parts regarding DSL implementing phases. They are (i) designing DSL, (ii) creating generator, (iii) using this new DSL for final solution.

The structure of DSL is clearly distributed in various models respectively aspects of each concepts as is described in Chapter 2. Every model has clear responsibility and usage. Therefore the user knows intuitively, which model should be used for each specific part of the DSL. The great advantage is possibility of two different views of the single definitions. Exploring definition for every concept in one model is great for overview. On the other hand, going through all aspects of one concept is excellent for checking of features of one specific concept.

The process of creating generator is quite complex. Understanding of using rules, especially weaving rules, macros and other opportunities of generator is not trivial. But with published MPS screencast tutorials it is possible. Existence of the base language greatly simplifies the development of extended language, because the user does not have to write the generator from concepts to text on his own. In contrary, template declaration must be a correct concepts. It is sometimes annoying, but comprehensible.

The interface for developing solutions is as powerful as other IDE like Eclipse or Visual Studio. However, the user has to get used to special style of editing as is described in Chapter 2. There are so called virtual packages, user can split his classes with them as with normal packages in Java. But the generated code ignores these virtual packages and is placed in one directory. This is sufficient for DSL, because it is not supposed to be included into any other IDE and edited directly. But we have considered it as restrictive for generated solution code. It would be helpful, if the generator adds this virtual package as suffix of class's package, which is acquired from model hierarchy.

For every part of developing is very important and beneficial the existence of already created DSL, samples and of course the base language. User can utilize them as a base for his extensions, directly in solution and as a teaching materials. These sources are either a part of MPS or can be downloaded from MPS's repository [12].

7.2.2 Used Part in RTEJ

The large part of MPS's functionality was tested in RTEJ, especially models Structure, Editor, Constraint and Generator. In these models we have used majority of offered features. But there are still many others, in the type system model we implemented only inference and non-typesystem rules. The text generator model was not used at all. Furthermore, there are dozens of mini languages implemented by MPS team, which are extending basic MPS with other features.

7.2.3 Deploying DSL

Nowadays there is no possibility to export created DSL to other IDE from MPS. Developers have promised, that in MPS version 3.0 there will be a support for deploying languages into IntelliJ IDEA as well as Eclipse [10]. IntelliJ IDEA is IDE for the Java language and is developed by JetBrains.

Nevertheless the programer has an opportunity to import DSL generated by another user of MPS. This import is rather simple, programer copies a jar file of the DSL into the directory with other languages in MPS's directory. It would be better, if there exists some integrated GUI in IDE to import the new DSL easier. But existed solution is sufficient and fulfills requirements for using DSL for other users than creator.

Once the jar file of DSL is placed in the correct directory, the user can utilize concepts of that DSL in his solution. There is a possibility to create a new solution without creating a new language. So user can establish a new project, import existed DSL without messing up the project viewer with the language model.

Building of DSL's jar file is rather simple. The user has to create the building model, there is the building script in this model. Settings of this script is already defined with default values, therefore the user has just to run it to build desired jar file.

7.2.4 Support

There exists official forum on Jetbrains's web-page [9]. The rate of response is quite good. Both MPS's developers and experienced users react on questions there. During working on this thesis the developers of MPS were contacted via email, they were very helpful.

Developers and experienced users publish some videos about usage of MPS [11]. The main contributors are Vaclav Pech and Markus Völter. These screencast videos are describing both basics about MPS and advanced themes. They are very beneficial for everybody.

There exist online documentation and reference material on MPS web-page. These sources are describing many parts of MPS including basic tutorials. But any complete documentation for all concepts does not exist, type-system rules, constraints etc. Thereby is rather complicated to decide, which element of MPS's huge possibilities should be chosen to achieve programer's goals. Especially the inline help of specific concepts in context menu is missing. Something similar to context menu generated from JavaDoc like in other IDE would be great. Therefore implementing of RTEJ in this thesis consumed a huge amount of time. With experience from implementing of RTEJ, the next DSL's development would take only a fragment of time.

MPS is still being developed. That is good, because the new features and fixes of bugs are released very often. On the other hand, the user must download and install the full MPS program for each new version. The size of MPS is about 200 MB, that is not much, but the patch system would be better. Fortunately settings of MPS and directory for user's projects are in user's home directory, so the user does not have to set anything after installing the new build.

During developing RTEJ some minor bugs were found. But they had been already reported in MPS's issue tracker and were fixed in a next build.

7.2.5 Conclusion

Jetbrains MPS is a powerful tool for creating new DSLs. Its possibilities for defining the structure, behavior and generator of DSL are great. But there are some problems, too. Especially the lack of inline documentation and restriction of using created DSL only in MPS. Despite these problems MPS is a good choice for DSL development. In addition, developers are still working on improving this IDE.

8. Related Work

8.1 Languages for RT Systems

This thesis is based on Realtime Specification for Java. But there are other languages, which are often used for developing realtime systems. In this section are briefly described the most famous ones

- C/C++ language - the first language for RT systems;
- Ada - language directly targeted for real-time and safety critical systems, this language has concurrency and real-time mechanism directly included;
- SCJ - Java framework for safety critical systems.

8.1.1 C/C++ Language

The C and C++ languages provide the programer a great deal of freedom, they are more technical and general than RTEJ. But that freedom is not the best way, when the real-time system should be created. C/C++ languages absolutely facilitate to create real-time system, especially with using *POSIX*. POSIX is a family of standards specified by IEEE. However, power of these languages enables huge opportunity for errors, therefore there are various subsets of these languages and programming styles to guarantee more safe developing of RT and SC systems. One of these standards is for example MISRA-C, which is a development standard for the C language in critical systems. Another is JSF C++, which is used for programming in C++ for avionics software and caused migration from Ada to C++ as main language of US Department of Defense.[13]

Besides of programming style and restrictions for certain statements, these standards define more exactly behavior of compilers, because specification of the C language is in some parts vague, especially working of generated code optimizer.

8.1.2 Ada

The first version of this language has been created in 1980s. Language has still been developed and nowadays the Ada 2012 is being prepared. The last stable release Ada 2005 is defined by ISO/ANSI standard. Ada is strongly typed language, block structured and targeted to embedded and realtime systems. It is member of Pascal languages family, so syntax is more verbal than for example the C language or Java[1].

Because Ada is determined for real-time and safety critical systems, is includes various static analysis to fix errors during writing the code. Furthermore strict syntax rules grant easier formal verification of application and compiler itself is validated for reliability in safety-critical applications.

Ada is a full-featured language, however, there exists so-called Ravenscar profile. This profile limits libraries and statements to meet requirements on determinism, schedulability analysis and memory selection. The Ravenscar profile is designed for small and efficient run-time systems.

8.1.3 Safety Critical Java

Safety Critical Java (SCJ) has been created as a framework for developing and analyzing programs for safety critical systems and their certification. Safety critical systems are systems, where failure could cause human death, extensive damage to property or harm the environment. Therefore these systems must be validated against certificates. Formal verification is difficult especially when program language provides too wide variability. That is the reason of creating SCJ. [8, 23]

SCJ has been build over the RTSJ by Java Community Process. RTSJ grants too much freedom for safety critical systems. Thereby SCJ uses only several features from RTSJ and installs additional limitations, thanks that the formal verification of created models is possible. Another advantage of this limitations is greater opportunity for static analysis during developing the code. These analysis are focused to decrease count of runtime exceptions by informing programer, which restrictions are broken, for example restrictions of referencing variables from different memories.

Programs are composed from *missions*, each mission is composed from bounded count of schedulable objects. There are three types of these objects:

- PeriodicEventHandler,
- AperiodicEventHandler,
- NoHeapRealtimeThread.

Each mission has its own bounded memory area, each schedulable objects has its own private part, too.

Types of this objects are restricted by the compliance level. There are three levels of compliance:

- Level 0 - the most restrictive level, known as a timeline model, a frame-based model, or a cyclic executive model. Only `PeriodicEventHandler` is supported as a type for schedule objects. `object.wait` and `object.notify` are not allowed.
- Level 1 - known as multitasking programming model, there is only one mission, which consists of concurrent computations with priority. `PeriodicEventHandler` and `AperiodicEventHandler` are enabled as a type for schedule objects.
- Level 2 - the most free level, program starts with one mission, but other missions could be created. All these missions run concurrently. All types of schedulable objects are enabled. Additionally, `object.wait` and `object.notify` could be used.

Besides of the compliance level, it is necessary to mention, that in SCJ it is not possible to access the heap. RTSJ has already reduced it by defining `NoHeapRealtimeThread`, but SCJ blocks heap for all structures.

Compared with RTEJ, SCJ is much more restrictive. On the other hand it is really suitable for safety critical system thanks to possibility of formal verification.

8.2 Modeling Languages

RTEJ is designed directly for implementation of a realtime system. However, there are languages for modeling these systems, both hardware and software part. SysML [14] and MARTE [15] number among these modeling methodologies. They are based on UML[1]. SysML is a general-purpose modeling language for systems engineering applications. It supports all part of application's design - analysis, design, verification and validation. MARTE targets directly realtime and embedded systems. Besides common parts of modeling language, it declares model's annotations with information, which enable specific analysis, especially performance and schedulability analysis.

To summarize, modeling methodologies are determined for defining a ways how to model, specify and describe an application, they do not serve for implementation as RTEJ.

8.3 RTSJ Framework and DSL

The problem with difficulty of programming in RTSJ, which is being solved in this thesis, is well known. There already exist several frameworks, which are trying to solve it. One of them is *SOLEIL* [18]. SOLEIL is a framework, which applies the Component Based Software Engineering (CBSE) paradigm into Real Time Specification for Java. The main goals are enhancement of development with RTSJ, increasing both efficiency through CBSE and generative programming, and brings safety through the formal verification. So SOLEIL is focused on the CBSE in contrast to RTEJ. Moreover it is much larger project, however main goals are similar.

8.4 DSLs Created in JetBrains MPS

JetBrain MPS is relatively new application, thereby there are not many created DSLs in this IDE. Majority of them are created by MPS developing team as a extension for the base language. Besides them there exists the project mbeddr [21]. This DSL is designed for developing embedded software based on the C language, created solution are developed in extended C language. Because that, mbedder's developers had to implement own base language, it also included write text generator. RTEJ uses the MPS's default base language, therefore this job was not necessary.

[1] Unified Modeling Language

9. Conclusion and Future Work

9.1 Summary of Work

In the beginning of thesis we have described RTSJ, we presented its power and reason of creating. On the other hand we have mentioned its weaknesses. The biggest ones are the difficulty of correct handling and prone to user's errors. The main goal of thesis was to design DSL, which removes or at least minimize these weaknesses. We named this new DSL as RealTime Extension for Java (RTEJ). We have created RTEJ as extension of Java. It means, that it adds new concepts to the set of regular concepts of the Java language.

As a first step we analyzed structure of this new DSL. The main part came from RTSJ, but there were done some restrictions of variability to improve user-friendly level and clearness of the whole solution. Besides of these basic code concepts the RTEJ contains also the complex ones. They serve for implementing of design patterns for real time system. These patterns have been explored and three of them have been chosen to be included into RTEJ.

We have decided to split all concepts in RTEJ in two groups. The first group is represented by statements used normally in code. The second one contains concepts on root level, that means the same level as file with class. The root concepts perfectly suit for various definitions of important structures like realtime threads or memory areas. During the whole DSL designing we had to balance the complexity for users, power of RTEJ and difficulty of implementation.

After finishing design we have implemented it in Jetbrains MPS. Once we had defined basic rules for structure, it became time for tuning various constraints, type systems, checks etc. This set of addition rules has improved work with RTEJ. The last but not least part of RTEJ is the Generator model, which responds for correct generation of code.

The second goal of this thesis was to evaluate JetBrains MPS. Testing of this IDE has been done by implementing of the RTEJ itself. By creating of this nontrivial DSL we have tested a great part of MPS's features. The power of MPS and its clearness is outstanding. We have not found any important feature for DSL, which could not be realized by this IDE. However, finding the way how to implement some features is quite difficult. It is caused by the lack of MPS documentation. There are although some basic tutorials and MPS team is producing screen-casts, but there are large parts of MPS, which are still without any documentation. Once this obstacle is overcome the creation of DSL will be pretty fast.

With every created DSL in MPS, the benefits of the whole system of MPS languages will be more significant. Because it is extremely easy to use already created languages for own solutions. There is already a huge amount of sub languages created by MPS team, which provides various features, e.g. table layout. As a result of this evaluation we have found JetBrains MPS as a proper tool for creating DSL's and language extensions.

Evaluation of created DSL was the last goal. It was necessary to verify, whether RTEJ is qualified to create a functional real-time system. This verification has been done by implementing the example - the sweet factory. Re-

implementation of this example with RTEJ was trouble-free. This test has shown, that RTEJ is suitable for implementing real-time system.

9.2 Future Work

Improvement of the data flow analyzer for memory area referencing would be beneficial, especially the correct handling with all possible references, even with objects returned by method calling.

Another problem which could be analyzed more deeply is related to local variables of RTEJ/RTSJ structures. We have described that problem in evaluation in Section 7.1.

RTEJ can be further improved. More MPS's intentions and various quick transformations of code can be added. Besides of these improvements of user's comfort there are still other design patterns, which are often used in real-time systems. Their implementing could be a non-trivial future extension.

More complex extension of RTEJ is adding possibility to generate either RTSJ or SCJ code. Besides of a switch in `MainDefinition` to define which output should be generate, various changes in concepts structure and especially in the code generator would be necessary. Moreover, new restrictions and concepts for control of SCJ compliance level would have to be added.

Bibliography

[1] ADA RESOURCE ASSOCIATION. *Ada information clearinghouse* [online]. [cit. 2012-03-12]. Available on: http://www.adaic.org/.

[2] BENOWITZ, E. G. and NIESSNER, A. F. A Patterns Catalog for RTSJ Software Designs. *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops* [online]. 2003, Volume 2889/2003, pp. 497-507. DOI: 10.1007/978-3-540-39962-9_55. Available on: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.3352.

[3] FOWLER, Martin. *Domain-Specific Languages*. 1st edition. Boston: Addison-Wesley Professional, 2010. 640 p. ISBN 978-0321712943.

[4] FOWLER, Martin. *Language Workbenches: The Killer-App for Domain Specific Languages?* [online]. 2008-01-09 [cit. 2011-09-05]. Available on: http://martinfowler.com/articles/languageWorkbench.html.

[5] IBM. *Real-time Java, Part 5: Writing and deploying real-time Java applications* [online]. [cit. 2011-09-15]. Available on: http://www.ibm.com/developerworks/java/library/j-rtj5/.

[6] JAVA COMMUNITY PROCESS. *Java Community Process* [online]. [cit. 2011-08-10]. Available on: http://jcp.org/en/jsr/summary?id=rtsj.

[7] JAVA COMMUNITY PROCESS. *Real-time specification for Java* [online]. [cit. 2011-08-10]. Available on: http://www.rtsj.org/.

[8] JAVA COMMUNITY PROCESS. *Safety Critical Java Technology Specification* [online]. [cit. 2012-03-20]. Available on: http://jcp.org/en/jsr/summary?id=302.

[9] JETBRAINS. *Develping forum for JetBrains MPS* [online]. [cit. 2011-11-29]. Available on: http://forum.jetbrains.com/forum/Meta-Programming-System.

[10] JETBRAINS. *JetBrains Meta Programming System* [online]. [cit. 2011-07-10]. Available on: http://www.jetbrains.com/mps/.

[11] JETBRAINS. *Screencast channel of JetBrains MPS* [online]. [cit. 2011-12-5]. Available on: http://tv.jetbrains.net/channel/mps.

[12] JETBRAINS. *Version repository of JetBrains MPS* [online]. [cit. 2011-11-10]. Available on: http://git.jetbrains.org/?p=mps/mps.git.

[13] OBILTSCHNIG, Günter. *C++ for Safety-Critical Systems* [online]. 2009-01-20 [cit. 2012-03-18]. Available on: http://www.appinf.com/download/SafetyCriticalC++.pdf.

[14] OBJECT MANAGEMENT GROUP. *OMG System Modeling Language specification*[online]. [cit. 2012-03-11]. Available on: http://www.omg.org/spec/SysML/.

[15] OBJECT MANAGEMENT GROUP. *The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems*[online]. [cit. 2012-03-10]. Available on: http://www.omg.org/spec/MARTE/.

[16] PIZLO, F., FOX, J. M., HOLMES, D. and Vitek, J. Real-Time Java Scoped Memory: Design Patterns and Semantics. *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* [online]. 2004, pp. 101-110. ISBN: 0-7695-2124-X. Available on: http://dx.doi.org/10.1109/ISORC.2004.1300335.

[17] PLŠEK, Aleš. *Real-time Java VMs* [online]. 2009-06-30 [cit. 2011-12-15]. Available on: http://rtjava.blogspot.com/2009/07/real-time-java-vms.html.

[18] PLŠEK, Aleš, Frederic Loiret, Michal Malohlava. *SOLEIL* [online]. [cit. 2012-02-20]. Available on: http://adam.lille.inria.fr/soleil/.

[19] TIMESYS. *RTSJ Reference Implementation and Technology Compatibility Kit* [online]. [cit. 2011-08-02]. Available on: http://www.timesys.com/java/.

[20] VÖLTER, Markus. Embedded Software Development with Projectional Language Workbenches. *Model Driven Engineering Languages and Systems 13th international conference*. Volume 6395/2010. Heidelberg: Springer-Verlag. Pages: 32-46. ISBN: 3642161286.

[21] VÖLTER, M., RATIU, D., THIEDE, M., MERKLE, B., KOLB, B., MATZAT, M. and PAVLETIC, D. *Embedded development using C language extensions* [online]. [cit. 2011-12-20]. Available on: http://mbeddr.wordpress.com/.

[22] WELLINGS, Andrew. *Concurrent and Real-Time Programming in Java*. 1st edition. Wiley, 2004. 446 p. ISBN 978-0470844373.

[23] ZHAO, L., TANG, D. and VITEK, J. A technology compatibility kit for safety critical Java. *JTRES '09 Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems* [online]. 2009, pp 160-168. ISBN: 978-1-60558-732-5. Available on: http://dl.acm.org/citation.cfm?id=1620428.

A. Content of Attached CD ROM

The thesis has attached CD ROM with binaries and source code of the created DSL and case-study.

`/README.TXT`
: Brief description of the content of the CD ROM.

`/doc/`
: Electronic version of this thesis.

`/rtej/`
: Source code of RTEJ represented as a MPS project.

  `/rtej/artifacts`
  : Compiled jar files, which can be included into MPS directory with other languages.

  `/rtej/language`
  : Files of RTEJ, both XML represantation and generated Java files.

  `/rtej/solutions`
  : Files of the sweet factory example. They included generated Java source code.

B. Installing of RTEJ

To develop a real time system by using RTEJ, there are needed three actions. Firstly download and install the last build of MPS from JetBrains's website[1]. There are distributions for Windows, Mac OS X and Linux. Because JetBrains MPS is open source, on the same website is possible to download the sources for project in IntelliJ IDEA and build MPS on himself.

After installing MPS it is necessary to place RTEJ's jar in the right place. Both jars must be placed in directory `%MPS installation dir%/languages`.

The last needed step is to create a new project in MPS, it does not have to contain a new language part. After creation of project, the solution properties must be opened and the RTEJ language added into `Dependecies/Used Languages`. After confirmation this action everything is prepared, this last step must be done for each solution.

[1] http://www.jetbrains.com/mps/download/index.html

C. More Complex Constraint

There is code of restrictions of memory in concept `MemoryAreaReference` in Figure C.1. This concept is used in various concepts and this constraint solves, which memory areas can be used in specific situation. Problem with local scoped areas is the most complex part of the presented code.

```
concepts constraints MemoryAreaReference {
  can be child <none>

  can be parent <none>

  can be ancestor <none>


  <<property constraints>>

  link {memory}
    referent set handler:<none>
    search scope:
      (model, scope, referenceNode, linkTarget, enclosingNode, operationContext)
        // constraint for limitation of scope of references to memory areas
        // show only global areas and local memories allocated in this class

        nlist<MemoryArea> result = new nlist<MemoryArea>;
        // class of this node
        node<ClassConcept> currentClass
            = enclosingNode.ancestor<concept = ClassConcept>;
        // all memory area in app
        nlist<MemoryArea> allArea = model.nodes(MemoryArea);
        // add global memory areas
        foreach specificArea in allArea {
          if (specificArea.global) {
            result.add(specificArea);
          }
        }
        // node is in class, not root one
        if (currentClass != null) {
          // is parrent field or static field declaration
          boolean isFieldDeclaration =
              enclosingNode.ancestor<concept = FieldDeclaration> != null ||
              enclosingNode.ancestor<concept = StaticFieldDeclaration> != null;
          // add local memory areas allocated in this class
          foreach specificArea in allArea {
            if (!(specificArea.global) &&
                specificArea.ancestor<concept = ClassConcept> ==
                currentClass) {
              // field declaration can reference only immediately alloceted
              if (!(isFieldDeclaration)
                  || ((node<ScopedMemory>) specificArea).immediateAlloc) {
                result.add(specificArea);
              }
            }
          }
        }

        // return found areas
        return result;
      }
    validator:
      <default>
    presentation :
      <no presentation>
  ;

  default scope
    <no default scope>

}
```

Figure C.1: Constraint aspect of concept MemoryAreaReference

D. List of Concepts

D.1 Root Package

MainDefinition extends `BaseConcept`
Main node with basic declaration about app.
Components:
`ClassConcept` mainPreInit
`ClassConcept` mainPostInit
`Mode` startingMode

D.2 Package `memory`

AllocateArray extends `Expression`
Allocate array in memory
Components:
`MemoryAreaReference` memory
`Type` allocater
`Expression` count

AllocateMemory extends `Statement`
Allocate new memory
Components:
`ScopedMemory` memory

AllocateVariable extends `Expression`
Allocate object in memory
Components:
`MemoryAreaReference` memory
`Type` allocater

ClassMemoryArea extends `NodeAttribute`
Components:
`ClassMemoryAreaType` type

ClassMemoryAreaType extends `integer`

EnterMemory extends `Statement`
Run code in memory
Components:
`MemoryAreaReference` memory
`Expression` runnable

ImmortalMemory extends `MemoryArea`
Immortal memory

MemoryArea extends `BaseConcept` implements *INamedConcept*
Abstract ancestor for memories
Components:
`boolean` global

MemoryAreaReference extends `BaseConcept`
Reference to memory
Components:
`MemoryArea` memory

ScopedMemory extends `MemoryArea`
Craete new ScopedMemory
Components:
`ScopedMemoryType` type
`boolean` wedgeThread
`boolean` immediateAlloc
`Expression` initial
`Expression` maximal

ScopedMemoryType extends `integer`

D.3 Package `memory.rawMemory`

RawMemory extends `BaseConcept` implements *INamedConcept*
Create access-point to raw memory
Components:
`Expression` base
`Expression` size

RawMemoryGet extends `Expression`
Get a value from raw memory
Components:
`RawMemoryType` varType
`RawMemory` rawMemory
`IntegerConstant` offset

RawMemorySet extends `Statement`
Set value in raw memory
Components:
`RawMemoryType` varType
`RawMemory` rawMemory
`IntegerConstant` offset
`Expression` value

RawMemoryType extends `integer`

D.4 Package `memory.sizeEstimator`

SizeEstimator extends `Statement` implements *INamedConcept*
Create new SizeEstimator

SizeEstimatorEstimate extends `Expression`
Estimate size of all objects reserved in this estimator
Components:
`SizeEstimator` sizeEstimator

SizeEstimatorReserve extends `Statement`
Reserve number of objects in estimator
Components:
`Type` objectType
`IntegerConstant` number
`SizeEstimator` sizeEstimator

D.5 Package `mode`

ChangeMode extends `Statement`
Change mode and execute all defined mode's actions
Components:
`Mode` mode

Mode extends `BaseConcept` implements *INamedConcept*
Define new mode
Components:
`RealtimeThreadReference` runThreads
`RealtimeThreadReference` stopThreads
`ModeChangePriority` changePriority
`ModeChangeRP` changeRP

ModeChangePriority extends `BaseConcept`
Change thread's priority
Components:
`IntegerConstant` priority
`RealtimeThread` thread

ModeChangeRP extends `BaseConcept`
Change thread's release parametres
Components:
`RealtimeThread` thread
`ReleaseParametres` releaseParametres

D.6 Package `pattern.channel`

CommunicationChannel extends `Statement` implements *INamedConcept*
Create new communication channel
Components:
`CommunicationChannelImplType` implType
`Type` messageType
`IntegerConstant` messageNumber

CommunicationChannelElement extends `Interface`
Interface for elements of communication channels

CommunicationChannelImplType extends `integer`

CommunicationChannelIsEmpty extends `Expression`
Return if is channel empty
Components:
`CommunicationChannel` channel

CommunicationChannelPop extends `Expression`
Pop message from the channel
Components:
`CommunicationChannel` channel

CommunicationChannelPush extends `Statement`
Push message to the channel
Components:
`Expression` object
`CommunicationChannel` channel

D.7 Package `pattern.objectPool`

ObjectPool extends `BaseConcept` implements *INamedConcept*
Create new object pool
Components:
`Type` type
`IntegerConstant` size
`BooleanConstant` expandable

ObjectPoolFree extends `Statement`
Free object created in the object pool
Components:
`Expression` object

ObjectPoolGet extends `Expression`
Get free object from the object pool
Components:
`ObjectPool` objectPool

D.8 Package `pattern.wedgeThread`

WedgeThreadStart extends `Statement`
Start job of the wedge thread
Components:
`ScopedMemory` memory

WedgeThreadStop extends `Statement`
Stop job of the wedge thread
Components:
`ScopedMemory` memory

D.9 Package `thread`

AllocateThread extends `Statement`
Allocate new thread and run it
Components:
`RealtimeThread` thread

AperiodicThread extends `RealtimeThread`
Create new aperiodic thread
Components:
`ReleaseParametresAperiodic` releaseParametres

InterruptThread extends `Statement`
Interrupt thread
Components:
`RealtimeThread` thread

PeriodicThread extends `RealtimeThread`
Create new periodic thread
Components:
`boolean` generateWaitCycle
`undefined` releaseParametres

RealtimeThread extends `Expression` implements *INamedConcept*
Default type of realtime thread
Components:
`boolean` noHeapThread
`boolean` startMain
`Expression` priority

`Expression` logic
`ReleaseParametres` releaseParametres
`MemoryArea` memory

RealtimeThreadReference extends `BaseConcept`
Reference for realtime thread
Components:
`RealtimeThread` thread

RunThread extends `Statement`
Run thread
Components:
`RealtimeThreadReference` thread

SporadicThread extends `RealtimeThread`
Create new speriodic thread
Components:
`undefined` releaseParametres

WaitForNextPeriod extends `Expression`
Wait for calling next thread's period, return boolean.

D.10 Package `thread.releaseParameter`

ReleaseParametres extends `BaseConcept`
Abstract ancestor for release parameters
Components:
`ClassConcept` overrunHandler
`ClassConcept` missHandler
`IntegerConstant` cost
`IntegerConstant` deadline

ReleaseParametresAperiodic extends `ReleaseParametres`
Release parameters for aperiodic threads

ReleaseParametresPeriodic extends `ReleaseParametres`
Release parameters for periodic threads
Components:
`IntegerConstant` startTime
`IntegerConstant` periodTime

ReleaseParametresSporadic extends `ReleaseParametres`
Release parameters for speriodic threads
Components:
`IntegerConstant` minInterval