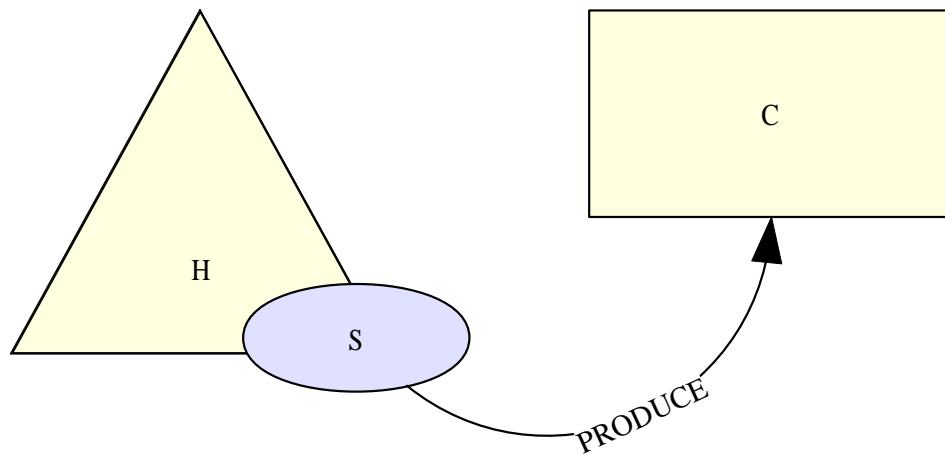


Verification of Java implementations against *ClassZ* specifications



Jonathan Owen Hugh Nicholson

Name: NICHOLSON, Jonathan Owen Hugh

Registration Number: 0531179

Login: johnic

Scheme: MSc Computer Science

Primary Supervisor: Prof. Ray Turner

Secondary Supervisor: Dr. Amnon Eden

Date: September 2006

Abstract

This project is aimed at those who wish to learn more about the verification of Java implementations against *ClassZ* specifications. It is not intended as an introduction or tutorial to *ClassZ* as that is outside the scope of this dissertation. It does however provide an account of the research undertaken to produce a working verification system for the TTPToolkit, and in depth discussions thereafter. All this leads to the conclusion that it is both possible to create such a package to a basic level, but there is much work to be done on both the current TTPToolkit and the verification implementation to accomplish more advanced features.

Acknowledgements

It has been a steep learning curve researching for this dissertation, and special thanks should be given to Dr. Amnon Eden and Prof. Ray Turner for giving me guidance and very important discussions on this topic which have lead to a much more focused and useful piece of research.

I would also like to thank Epameinondas Gasparis for teaching me a great deal about *ClassZ* and delaying his own study to provide me with both interesting discussions and modifications to the TTPToolkit to allow me to do my work.

Perhaps one of the most fun parts of producing this dissertation is that it is contributing to ongoing research and that I have had the support of my peers, not only those working on similar projects, but also those working on completely different research. I'm sure I could not have produced what I have without our late night, coffee/tea fueled stints in the labs.

Finally I'd like to thank, and dedicate, this piece of research to my parents, who have always believed that I can push myself beyond my limitations, and have always given me the opportunities to do so.

Contents

Abstract	i
Acknowledgements	ii
Contents	iii
List of Tables	v
List of Figures	vi
Glossary of terms	vii
Mathematical symbols	viii
Introduction and motivation	1
Literature review	2
Aims and Objectives	3
Aim	3
Objectives	3
Style conventions	3
I Theory	4
1 Investigation into <i>ClassZ</i> and verification	5
2 Mathematical verification	7
2.1 Validation rules	8
2.1.1 Existence	8
2.1.2 Impositions	9
2.1.3 Relations	9
2.2 Mathematical statements for Correctness	10
2.2.1 Unary relations	10
2.2.2 Binary relations	10
2.3 Pseudo transitive closure correctness equations	14
2.3.1 Proof of pseudo transitive closures	15
3 Verification examples	16
3.1 Extensional verification case study	17
3.1.1 Specification (Ψ)	17
3.1.2 Program code	17

3.1.3	Pass	18
3.1.4	Fail	19
3.2	Intensional verification case study	20
3.2.1	Specification (Ψ)	20
3.2.2	Program code and Design Model (\mathcal{M})	20
3.2.3	Discussion:	20
3.3	Extreme inheritance case study	21
3.3.1	Specification (Ψ)	21
3.3.2	Pass	21
3.3.3	Fail	22
3.3.4	Discussion	22
II	Package Development	24
4	Specification	25
4.1	Functional requirements	26
4.1.1	Must have	26
4.1.2	Nice to have	27
4.1.3	Features for the next version	27
4.2	Non-functional requirements	28
5	Design	29
5.1	The verification algorithm	30
5.2	Design Policy	34
5.3	Package dependencies	35
5.4	Package design	36
5.5	Interface to the verifier package in the TTPtoolkit	38
6	Testing	39
6.1	Testing strategy	40
6.2	Discussion of code inspection	40
6.3	Discussion of test case results	41
7	Evaluation of the Implementation	44
7.1	Limitations imposed on the Verification package	44
7.2	Limitations of the Verification package	47
7.3	Successes of the Verification package	48
7.4	Advancements to the Verification package	49
8	Conclusions	50
	Bibliography	52
	Appendices	53
Appendix A	List of test cases	54
Appendix B	Results of listed Test Cases	64
Appendix C	Proposed <i>ClassZ</i> DTD file	69
Appendix D	Possible bugs found in the TTPtoolkit V0.3	70
Appendix E	Graphical User Interface	71
Appendix F	CD Contents (including source code)	74

List of Tables

2.1	Possible domains of the known unary relations	10
2.2	The domains and ranges of the known binary relations	10
2.3	Correctness of the <i>CALL</i> relation	11
2.4	Correctness of the <i>INVOKE</i> relation	11
2.5	Correctness of the <i>CREATE</i> relation	11
2.6	Correctness of the <i>RETURN</i> relation	12
2.7	Correctness of the <i>PRODUCE</i> relation	12
2.8	Correctness of the <i>INHERIT</i> relation	12
2.9	Correctness of the <i>MEMBER</i> relation	13
2.10	Correctness of the <i>MEMBERS</i> relation	13
2.11	Pseudo transitive closure correctness equations	14
5.1	Interface commands to the ttp.verifier package	38

List of Figures

3.1	Extensional specification of the Factory Method pattern	17
3.2	Intensional specification of the Factory Method pattern	20
3.3	Extensional specification of an extreme inheritance case	21
4.1	UML use case diagram for the ttp.verifier package	26
5.1	Flowchart showing an abstraction of the Existence rules in the verification algorithm	31
5.2	Flowchart showing an abstraction of the Imposition rules in the verification algorithm	32
5.3	Flowchart showing an abstraction of the Relation rules in the verification algorithm	33
5.4	Package dependencies for the ttp.verifier package	35
5.5	Package dependencies for the ttp.verifier package in UML	35
5.6	<i>ClassZ</i> diagram showing the verifier package	36
5.7	Detailed <i>ClassZ</i> diagram showing the verification of relations	37
7.1	Problems with access modifiers: Specification	46
7.2	Problems with access modifiers: Implementation	46
7.3	A case of failure for an Isomorphic INHERIT relation	48
8.1	<i>ClassZ</i> diagram showing the verifier package including the <i>THROWS</i> relation	51

Glossary of terms

Architectural Description Languages (ADL)

Top-down language used to specify architectural abstractions, otherwise known as non-functional requirements[EKF03]

Bloated

(or bloatware) An implementation that is larger than it needs to be, either in redundant functionality or the resources it requires.

Cardinality

The number of entities in any given set, commonly the # symbol is used for this function

ClassZ

The Second-Order language derived from LePUS, the topic of Dr. Eden's Phd thesis

First-Order Language

The language used to implement a program, in the context of this report this is Java

Formal Specification Languages (FSL)

Top-down language used to specify functional requirements[EKF03]

Object Oriented (OO)

A structure for things such as programming languages and databases. Data and behavioral logic is separated into classes (objects). Object Oriented systems also allow for multiple and/or single inheritance

Second-Order Language

The language used to represent a specification, in the context of this report this is *ClassZ*

Software Crisis

The term used to describe the complexity of writing correct, valid and understandable software as the environment for which it is designed for is rapidly modified.

Two-Tier Programming (TTP)

The philosophy that a specification should be written in a formal, mathematically based, language with emphasis on abstraction, both from any singular implementation language and to reduce the inclusion of unnecessary information into the specification.

Mathematical symbols

- such that, it seems some sources use a comma symbol ($,$) interchangeably for this. This is the symbol i am used to, therefor it is the one which is found throughout this dissertation.
- $\exists!$ There exists one and only one, for example $\exists!x \in \mathbb{Z} \bullet x = 5$ means that there exists only one value of x in the set of all integers which is equal to 5
- \exists There exists, for example $\exists x \in \mathbb{Z} \bullet |x| = 5$ means that there exists a value of x in the set of all integers whose absolute value is equal to 5.
- \forall For all, for example $\forall x \in \mathbb{Z} \bullet x = x$ means that for all integers x , x is equal to its self
- \Leftrightarrow Equality of implication, for example $X \Leftrightarrow Y$ means that X is satisfied by, and implies Y , and that Y is satisfied by, and implies X
- \mathbb{R} The set of relations in \mathfrak{M}
- \mathbb{U}_* The set of all entities of any dimension in \mathfrak{M} such that $\mathbb{U}_* = \mathbb{U}_0 \uplus \dots \uplus \mathbb{U}_d$, where \mathbb{U}_0 is the set of all ground entities, and $\mathbb{U}_d = \mathcal{P}^d(\mathbb{U}_0)$ is a subset of the power set in dimension d of \mathbb{U}_0
- \mathcal{R} A relation
- \mathfrak{M} The Design Model, where $\mathfrak{M} = \langle \mathbb{U}_*, \mathbb{R}, \otimes \rangle$
- \models Implements, or satisfies. For example $\mathfrak{M} \models \Psi$ says that the specification Ψ is satisfied by the Design Model \mathfrak{M}
- \otimes Superimposition operator, used in *ClassZ* as a binary operator between a signature and a class such that $S \otimes C$, means S is superimposed on C . See the theory section for details of how this is validated.
- Ψ A Specification
- \Rightarrow Satisfies or implies, for example $X \Rightarrow Y$ means that X is satisfied by, and implies Y
- \rightarrow If ... then ... operator, such that $x \rightarrow y$ means if x then y .

Introduction and motivation

The *Software Crisis* is a very serious problem for those who produce large software systems. The ad-hoc methods of software development employed by some companies and individuals are not acceptable in the long term. In the 1960's, when computing power available to the software engineer was at a minimum, systems had to be very carefully designed and implemented so as not become *bloated*. As computing power increases the software engineer has reduced hardware limitations, allowing more complex systems to be developed, with more choice in their design. Having this choice is by no means a bad thing; but inadequate knowledge, bad decisions and the actions thereafter can lead to a derogatory effect on the sustainability of the system lifecycle. Not only during the maintenance phase, but possibly stopping development altogether before the production of a release version.

”... instead of finding ourselves in a state of eternal bliss with all programming problems solved, we found ourselves up to our necks in the software crisis!”[Dij72, p.860]

The erosion in quality of developed systems can occur at any point during its lifecycle. Badly specifying requirements (London Ambulance Computer Aided Dispatch system[Fin]) and incorrect/falsified test results (THERAC25 case[Baa02][p.149]) are just two examples of how bad practice of software development can introduce possibly life threatening problems into a system.

Many methods and strategies have been created to reduce these issues, Architectural Description Languages (ADL) and Formal Specification Languages (FSL) for example. However none have specifically targeted the issue of architectural design, which is an interesting and current area of study. Therefore this dissertation will investigate *ClassZ*[Edeb], a language adhering to the Two-Tier Programming (TTP) philosophy, and the verification of specifications thereafter.

ClassZ is a *Second-Order Language* which allows the software engineer to easily represent large and complex system architectures in a formal way. Namely this is accomplished with charts and schemas, which provide a clear level of abstraction from implementation specific information, known as a *First-Order Language*.

The project is certainly an interesting area of research, one which I am proud to be a part of. However this dissertation should not be considered an introduction to *ClassZ* or the TTP philosophy. It provides in depth discussion on the mathematical and semantic properties of *ClassZ* in respects to verification, knowledge of *ClassZ* and a background in mathematics is assumed.

Literature review

This area of research is currently active, and although there are technical reports, previous dissertations, and a draft book being produced in this area, there is very little information available on the subject of verification. Books that are available on this matter are aimed at verifying compiled code to its implementation language. Often these titles discuss the use of Hoare's logic, which is beyond the needs of this type of verification.

There is however an MSc dissertation discussing verification of LePUS[Iya03], the predecessor to *ClassZ*, which has been examined and its relevance estimated in respects to what this project. It was decided that as this resource does not contain enough current, usable information for it to be used as a basis for this dissertation.

This work will be adding to the research being carried out by Dr. Amnon Eden, Prof. Ray Turner and Epameinondas Gasparis; also my peers Omololu Ayodeji, Dimitrios Fragkos, and Yi Xu. Previous work has been produced by Olumide Iyaniwura[Iya03], Gu Bo[Bo04] and Maple Liang[Lia04a, Lia04b].

Aims and Objectives

Aim

Investigate *ClassZ* to a sufficient level of understanding to the ends of implementing an operational verification package within the scope of the TTP toolkit.

Objectives

1. Research relevant subject areas, such as *ClassZ*, TTP, and software verification.
2. Design and develop the *ClassZ* verification package within the TTPtoolkit.
3. Finally to report on research findings and evaluate the developed system.

Style conventions

For the purposes of this report all *ClassZ* relations appear uppercase in the following style, for example a Total *INHERIT* relation would appear as:

Total(INHERIT, domain, range)

Inline code appears in upper or lowercase as the language syntax dictates, using the standard \LaTeX `verbatim/verb` environments in the following style:

`Code style`

The exception to this is when large sections of code are included, such as an entire class, in which case it appears in the following style:

```
Code style
```

This is processed using LGrind, a source code pretty printer for \LaTeX , available from the CTAN archives. Generally the code found in this document is of the Java language. However occasionally other languages, such as XML or DTD files, have used the same formatting for simplicity. It should be assumed any code is Java unless stated otherwise.

Part I

Theory

Chapter 1

Investigation into *ClassZ* and verification

”To verify a program means to prove that a program is consistent with its specifications. Subsequently the term ’Verification’ is used as a technical term referring to the process of proving consistency with specifications”[[Pol81](#)][p.2]

Before discussing *ClassZ* or verification, we must clarify exactly what a specification means from the point of view of the TTP philosophy. A specification provides enough positive information such that an implementation can be produced in accordance to it. This means that if a specification states that a class must be abstract, then it must be abstract, however if no such rule is included then the said class may or may not be abstract. In accordance to this we can say that the absence of information in the specification does not indicate the absence of information in the implementation.

”verification can be weak and need not (and generally does not) express all requirements for a program”[Pol81][p.3]

For the purposes of this dissertation verification is the process of generating a proof from a given *ClassZ* specification and validating an implementation in accordance to that proof. Verification is not validating the syntactic properties of a given implementation, this is left to the first-order language’s compiler to deal with. Nor does it attempt to validate the semantic properties of an implementation, as this is part of the design process and there are other more relevant forms of testing to this end. The current TTPToolkit implementation satisfies these needs by populating the Finite Structure using Java *.class files (Java Reflection), rather than parsing the source code found in the *.java files.

”... verification should not be confused with correctness in the intuitive sense (ie the program does what you expect it to do). Verification is not a substitute for other software engineering techniques such as systematic program development, testing, walk-through and so on; rather verification augments these techniques. It gives us an additional level of confidence that our program has the property expressed in its specification.”[Pol81][p.3]

Chapter 2

Mathematical verification

”Lacking evidence of a positive does not prove a negative, but rather might suggest a possibility of a negative. Numerous reproducible negative findings may enhance the probability of a negative”[[Nic](#)]

This chapter could be considered the core of this dissertation and requires some knowledge and understanding of mathematics and logic, definitions of symbols can be found in the ”Mathematical symbols” chapter at the start of this report. A lot of this material has been produced while heavily referencing a mathematical text book[[Epp95](#)]

2.1 Validation rules

These rules are summaries of those stated in Dr. Eden's draft book[Edea][p.78], and are not intended to be seen as replacements. Throughout these rules these terms are used:

C^d A class constant C of dimension d .

H^d A hierarchy constant H of dimension d , where $d > 0$. It should be noted that the symbol H is only used for the existence equations, beyond that terms C is used to include hierarchies as they are more specific types of classes.

M^d A method constant M of dimension d , this exists only in the Design Model as no notion of methods exist in a specification.

t, t_1, t_2 Constant terms, where (for example) $t_x^d = C^d \vee H^d \vee S^d$

One of the goals when creating the verification package is to ensure that these mathematical rules must be correctly implemented. If they are not then any verification results produced would be useless.

2.1.1 Existence

Classes: $C^d \in \Psi \rightarrow C^d \in \mathbb{U}_*$

Signatures: $S^d \in \Psi \rightarrow S^d \in \mathbb{U}_*$

Hierarchies: $H^d \in \Psi \rightarrow (H^d \in \mathbb{U}_* \wedge \forall h^1 \in H^d \bullet \exists root \in h^1 \bullet$

$\forall nonroot \in h^1 \wedge root \neq nonroot \bullet INHERIT(nonroot, root) \in \mathbb{R})$

It can be seen here that verification of hierarchies can be achieved in two steps. The first is to recursively create a set containing all hierarchies of dimension 1 that a given higher dimensional hierarchy contains. If the hierarchy is of dimension 1 already then the set contains a singleton of the given hierarchy. Each entity in this set can be checked for the properties of a hierarchy[Edea], such that there exists a root class which all other classes inherit from. Currently the draft[Edea] states that the root class must be abstract, however I believe this has been changed so no mention of it is made here.

2.1.2 Impositions

Clans: $S^0 \otimes C^d \in \Psi \rightarrow M^d \in \mathbb{U}_* \wedge$

$SIGNATUREOF(S^0, M^d) \in \mathbb{R} \wedge$

$MEMBER(M^d, C^d) \in \mathbb{R}$

Tribes: $S^1 \otimes C^d \in \Psi \rightarrow M^{d+1} \in \mathbb{U}_* \wedge SIGNATUREOF(S^1, M^{d+1}) \in \mathbb{R} \wedge \forall S \in S^1 \bullet$

$[M^d \in \mathbb{U}_* \wedge SIGNATUREOF(S, M^d) \in \mathbb{R} \wedge$

$MEMBER(M^d, C^d) \in \mathbb{R}]$

Logically, and with the aid of these equations if required, a Tribe can be considered a set of Clans; therefore they should be validated as such.

2.1.3 Relations

Unary: $\mathcal{R}(t^d) \in \Psi \rightarrow \mathcal{R}(t^d) \in \mathbb{R} \wedge \forall t_1^0 \in t^d \bullet \mathcal{R}(t_1^0) \in \mathbb{R}$

Ground: $\mathcal{R}(t^0, t_1^0) \in \Psi \rightarrow \mathcal{R}(t^0, t_1^0) \in \mathbb{R}$

Total: $Total(\mathcal{R}, t^a, t_1^b) \in \Psi \rightarrow$

$(a = 0 \wedge b = 0 \rightarrow \mathcal{R}(t^a, t_1^b) \in \mathbb{R}) \vee$

$(a = 0 \rightarrow Total(\mathcal{R}, \{t^a\}, t_1^b) \in \mathbb{R}) \vee$

$(b = 0 \rightarrow Total(\mathcal{R}, t^a, \{t_1^b\}) \in \mathbb{R}) \vee$

$(\forall x \in t^a \bullet ABSTRACT(x) \notin \mathbb{R} \wedge \exists y \in t_1^b \bullet Total(\mathcal{R}, x, y) \in \mathbb{R})$

Isomorphic: $Isomorphic(\mathcal{R}, t^a, t_1^b) \in \Psi \rightarrow$

$(a = 0 \wedge b = 0 \rightarrow \mathcal{R}(t^a, t_1^b) \in \mathbb{R}) \vee$

$(a = 0 \rightarrow Total(\mathcal{R}, \{t^a\}, t_1^b) \in \mathbb{R}) \vee$

$(b = 0 \rightarrow Total(\mathcal{R}, t^a, \{t_1^b\}) \in \mathbb{R}) \vee$

$((\forall x \in t^a \bullet ABSTRACT(x) \notin \mathbb{R} \wedge \exists! y \in t_1^b \bullet$

$ABSTRACT(x) \notin \mathbb{R} \wedge Isomorphic(\mathcal{R}, x, y) \in \mathbb{R})$

\wedge

$(\forall y \in t_1^b \bullet ABSTRACT(x) \notin \mathbb{R} \wedge \exists! x \in t^a \bullet$

$ABSTRACT(x) \notin \mathbb{R} \wedge Isomorphic(\mathcal{R}, x, y) \in \mathbb{R}))$

As seen in these equations, and as stated in Dr. Eden’s draft[Edea], there are cases in while Isomorphic relations should be validated as Total or ground relations, and similarly some cases where Total should be validated as a ground relation.

2.2 Mathematical statements for Correctness

To accompany the rules given above, the following sections outline transitivity and equivalence of the known unary and binary relations in respects to the Java language. These may or may not hold true for all *First-Order Languages*, but this dissertation is focused only on Java.

2.2.1 Unary relations

The following table indicates the possible domains of currently used unary relations, where the only relation of interest here is *ABSTRACT*. This is because *METACLASS* is not defined in the Java language and therefore cannot be validated. As such it will be ignored throughout the rest of the dissertation.

Domain	Relations
<i>CLASS</i>	<i>ABSTRACT, METACLASS</i>
<i>SIGNATURE</i>	<i>ABSTRACT</i>

Table 2.1: Possible domains of the known unary relations

2.2.2 Binary relations

Before proceeding into the mathematical and semantic properties of each relation, the possible range(s) and domain(s) of each must be Identified. Without this information it is impossible to be able to define specific equations based on their individual roles. Logically, and in respects to the Java language, the following table has been constructed to represent this information.

Domain	Range	Relations
<i>CLASS</i>	<i>CLASS</i>	<i>INHERIT, MEMBER, MEMBERS, CREATE</i>
<i>CLASS</i>	<i>SIGNATURE</i>	<i>CALL, INVOKE</i>
<i>SIGNATURE</i>	<i>CLASS</i>	<i>PRODUCE, CREATE, RETURN</i>
<i>SIGNATURE</i>	<i>SIGNATURE</i>	<i>CALL, INVOKE</i>

Table 2.2: The domains and ranges of the known binary relations

The validation of binary relations is a little more complex than unary relations. The following

is a set of correctness expressions, where ground relations are used as examples. However these rules also hold true for total/isomorphic predicates as long as every relation in that rule uses the same predicate. By using a combination of these unary and binary correctness equations, a strong mathematical proof can be created for a given specification.

Correctness of the *CALL* relation

$$\begin{aligned} CALL(dom, ran) &\Leftrightarrow INVOKE(dom, ran) \\ CALL^+(dom, ran) &\Leftrightarrow INVOKE^+(dom, ran) \end{aligned}$$

Table 2.3: Correctness of the *CALL* relation

This says that *CALL* relations are logically equivalent to *INVOKE* relations with the same *domain* and *range* values. However only the *INVOKE* relation exists in the current Design Model, so the use of these correctness equations should be to convert any uses of the *CALL* relation in a given specification to an *INVOKE* relation.

Correctness of the *INVOKE* relation

$$\begin{aligned} INVOKE^+(dom, ran) &\Leftrightarrow INVOKE(dom, ran) \\ &\quad \vee \\ &\quad [INVOKE(dom, x) \wedge INVOKE^+(x, ran)] \end{aligned}$$

Table 2.4: Correctness of the *INVOKE* relation

Correctness of the *CREATE* relation

$$\begin{aligned} CREATE^+(dom, ran) &\Leftrightarrow CREATE(dom, ran) \\ CREATE(dom, ran) &\Leftrightarrow \exists x \bullet INHERIT^+(x, ran) \wedge CREATE(dom, x) \end{aligned}$$

Table 2.5: Correctness of the *CREATE* relation

A *CREATE* relation also implies the invocation of a constructor of a given class. Currently in *ClassZ* there is no explicit way to define a constructor as there is with abstract, and there may never be due to the premise of *ClassZ* being based on the notion of abstraction.

Correctness of the *RETURN* relation

$$\begin{aligned} RETURN^+(dom, ran) &\Leftrightarrow RETURN(dom, ran) \\ RETURN(dom, ran) &\Leftrightarrow \exists x \bullet INHERIT^+(x, ran) \wedge RETURN(dom, x) \end{aligned}$$

Table 2.6: Correctness of the *RETURN* relation

Correctness of the *PRODUCE* relation

$$\begin{aligned} PRODUCE(dom, ran) &\Rightarrow CREATE(dom, ran) \\ PRODUCE(dom, ran) &\Rightarrow RETURN(dom, ran) \\ PRODUCE(dom, ran) &\Leftrightarrow CREATE(dom, ran) \wedge RETURN(dom, ran) \\ PRODUCE(dom, ran) &\Rightarrow \exists x \bullet INHERIT^+(x, ran) \wedge PRODUCE(dom, x) \\ PRODUCE^+(dom, ran) &\Leftrightarrow PRODUCE(dom, ran) \\ &\quad \vee \\ &\quad CREATE^+(dom, ran) \wedge RETURN^+(dom, ran) \end{aligned}$$

Table 2.7: Correctness of the *PRODUCE* relation

Correctness of the *INHERIT* relation

$$\begin{aligned} INHERIT^+(dom, ran) &\Leftrightarrow INHERIT(dom, ran) \\ &\quad \vee \\ &\quad [INHERIT(dom, x) \wedge INHERIT^+(x, ran)] \end{aligned}$$

Table 2.8: Correctness of the *INHERIT* relation

Correctness of the *MEMBER* relation

$$\begin{array}{l}
 MEMBER(dom, ran) \Rightarrow \exists x \bullet INHERIT^+(x, ran) \wedge MEMBER(dom, x) \\
 MEMBER^+(dom, ran) \Leftrightarrow MEMBER(dom, ran) \\
 \quad \vee \\
 \quad [MEMBER(dom, x) \wedge MEMBER^+(x, ran)]
 \end{array}$$

Table 2.9: Correctness of the *MEMBER* relation

Correctness of the *MEMBERS* relation

$$\begin{array}{l}
 MEMBERS(dom, ran) \Rightarrow MEMBER(dom, ran) \\
 MEMBERS^+(dom, ran) \Leftrightarrow MEMBERS(dom, ran) \\
 \quad \vee \\
 \quad [MEMBER(dom, x) \wedge MEMBERS^+(x, ran)] \\
 \quad \vee \\
 \quad [MEMBERS(dom, x) \wedge MEMBER^+(x, ran)] \\
 \quad \vee \\
 \quad [MEMBERS(dom, x) \wedge MEMBERS^+(x, ran)]
 \end{array}$$

Table 2.10: Correctness of the *MEMBERS* relation

2.3 Pseudo transitive closure correctness equations

In the equations given above some transitive relations defined to be equivalent to their non-transitive relation. Transitive closure of relation $\mathcal{R}(dom, ran)$, written as $\mathcal{R}^+(dom, ran)$, is defined as:

$$\bullet \mathcal{R}(dom, ran) \in \mathbb{R} \vee [\exists x \in \mathbb{U}_* \bullet \mathcal{R}(dom, x) \in \mathbb{R} \wedge \mathcal{R}^+(x, ran) \in \mathbb{R}]$$

Which works for relations such as *INHERIT* and *INVOKE*, but not all relations. Examine the *PRODUCE*⁺ relation, in accordance with the definition of transitive closure the following example is possible:

$$PRODUCE^+(dom, ran) = PRODUCE(dom, x) \wedge PRODUCE(x, ran)$$

Which is correct, however it may not be what the user intended to say. In other words the semantics of this relation may not match the mathematical definition. There is also the issue that the domain of a *PRODUCE* relations should always be a class entity, and its domain a method entity. In this example the second *PRODUCE* relation requires that its domain is a class entity, which cannot be accomplished in the Java language as a class does not have the ability to return anything. This is also true for other relations such as *CREATE*⁺ and *RETURN*⁺.

In an attempt to overcome this the following equations have been defined for discussion. They do not satisfy the mathematical definition of transitivity, and as such will not be part of the verification package. For these equations to be accepted as part of the *ClassZ* language the ⁺ operator would have to be redefined to allow both homogeneous and heterogeneous transitivity.

$$\begin{array}{l} CREATE^+(dom, ran) \Leftrightarrow CREATE(dom, ran) \\ \quad \vee \\ \quad [\exists x \in CALL(dom, x) \bullet CREATE^+(x, ran)] \\ RETURN^+(dom, ran) \Leftrightarrow RETURN(dom, ran) \\ \quad \vee \\ \quad [RETURN(dom, ran) \wedge \exists x \in CALL(dom, x) \bullet \\ \quad \quad RETURN^+(x, ran)] \\ PRODUCE^+(dom, ran) \Leftrightarrow PRODUCE(dom, ran) \\ \quad \vee \\ \quad CREATE^+(dom, ran) \wedge RETURN^+(dom, ran) \\ \quad \vee \\ \quad [RETURN(dom, ran) \wedge \exists x \in CALL(dom, x) \bullet \\ \quad \quad (PRODUCE^+(x, ran) \wedge RETURN(x, ran))] \end{array}$$

Table 2.11: Pseudo transitive closure correctness equations

2.3.1 Proof of pseudo transitive closures

This section is provided as a proof that the given equations in the above section are logically correct.

1. Starting with an example use of the $PRODUCE^+$ relation as defined above:

$$PRODUCE^+(dom, ran) \Leftrightarrow RETURN(dom, ran) \wedge CALL(dom, x) \wedge CALL(x, y) \wedge PRODUCE(y, ran) \wedge RETURN(y, ran) \wedge RETURN(x, ran)$$

2. Using the following rule:

$$PRODUCE(dom, ran) \Leftrightarrow CREATE(dom, ran) \wedge RETURN(dom, ran)$$

This can be rewritten as:

$$PRODUCE^+(dom, ran) \Leftrightarrow RETURN(dom, ran) \wedge CALL(dom, x) \wedge CALL(x, y) \wedge CREATE(y, ran) \wedge RETURN(y, ran) \wedge RETURN(y, ran) \wedge RETURN(x, ran)$$

And simplified by removing duplicates to:

$$PRODUCE^+(dom, ran) \Leftrightarrow RETURN(dom, ran) \wedge CALL(dom, x) \wedge CALL(x, y) \wedge CREATE(y, ran) \wedge RETURN(y, ran) \wedge RETURN(x, ran)$$

3. Using the rules:

$$RETURN^+(dom, ran) \Leftrightarrow RETURN(dom, ran) \vee [RETURN(dom, ran) \wedge \exists x \in CALL(dom, x) \bullet RETURN^+(x, ran)]$$

And:

$$CREATE^+(dom, ran) \Leftrightarrow CREATE(dom, ran) \vee [\exists x \in CALL(dom, x) \bullet CREATE^+(x, ran)]$$

This can be simplified to:

$$PRODUCE^+(dom, ran) \Leftrightarrow CREATE^+(dom, ran) \wedge RETURN^+(dom, ran)$$

4. Using:

$$PRODUCE(dom, ran) \Leftrightarrow CREATE(dom, ran) \wedge RETURN(dom, ran)$$

It can be shown that this can be further simplified to:

$$PRODUCE^+(dom, ran) \Leftrightarrow PRODUCE^+(dom, ran)$$

Which is correct.

Chapter 3

Verification examples

”We learn by example and by direct experience because there are real limits to the adequacy of verbal instruction.”

Malcolm Gladwell 2005[[quo](#)]

Examples are a great way to further our understanding of the verification process, as they pull together all the knowledge gained from study and logic.

3.1 Extensional verification case study

3.1.1 Specification (Ψ)

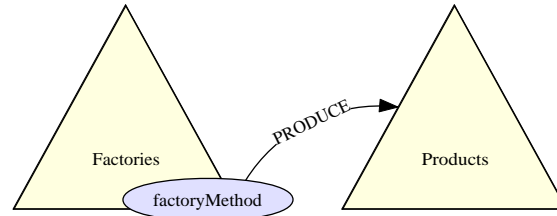


Figure 3.1: Extensional specification of the Factory Method pattern

3.1.2 Program code

Both examples use the program outlined in this very basic implementation of the Factory Pattern.

```

package factoryPattern;

public abstract class AbstractFactory {
    public abstract AbstractProduct factoryMethod();
5 }

```

```

package factoryPattern;

public class FactoryA extends AbstractFactory {
    public AbstractProduct factoryMethod()
5 { return new ProductA (); }
}

```

```

package factoryPattern;

public class FactoryB extends AbstractFactory {
    public AbstractProduct factoryMethod()
5 { return new ProductB (); }
}

```

```

package factoryPattern;

public abstract class AbstractProduct { }

```

```

package factoryPattern;

public class ProductA extends AbstractProduct{ }

```

```

package factoryPattern;

public class ProductB extends AbstractProduct{ }

```

3.1.3 Pass

Design Model (\mathfrak{M})

0 Dim classes: AbstractFactory, FactoryA, FactoryB, AbstractProduct, ProductA, ProductB

0 Dim signatures: factoryMethod()

0 Dim methods: AbstractFactory.factoryMethod, FactoryA.factoryMethod, FactoryB.factoryMethod

1 Dim classes: FactoryClasses = {FactoryA, FactoryB}
ProductClasses = {ProductA, ProductB}

1 Dim hierarchies: Factories = {AbstractFactory, FactoryClasses}
Products = {AbstractProduct, ProductClasses}

Unary relations: *ABSTRACT* = {AbstractFactory, AbstractProduct, AbstractFactory.factoryMethod}

Binary relations: *INHERIT* = {(FactoryA, AbstractFactory), (FactoryB, AbstractFactory), (ProductA, AbstractProduct), (ProductB, AbstractProduct)}
SIGNATUREOF = { (factoryMethod(), AbstractFactory.factoryMethod), (factoryMethod(), FactoryA.factoryMethod), (factoryMethod(), FactoryB.factoryMethod)}
MEMBER = { (AbstractFactory.factoryMethod, AbstractFactory), (FactoryA.factoryMethod, FactoryA), (FactoryB.factoryMethod, FactoryB)}
PRODUCE = {(FactoryA.factoryMethod, ProductA), (FactoryB.factoryMethod, ProductB)}

Explanation:

This implementation would pass verification as everything indicated in the specification is included in the Design Model, the required relations exist, and where relevant the Design Model also complies to the definition of Hierarchies.

3.1.4 Fail

Design Model (\mathcal{M})

0 Dim classes: `AbstractFactory, FactoryA, FactoryB, AbstractProduct, ProductA, ProductB`
 0 Dim signatures: `factoryMethod()`
 0 Dim methods: `AbstractFactory.factoryMethod, FactoryA.factoryMethod, FactoryB.factoryMethod`
 Unary relations: $ABSTRACT = \{AbstractFactory, AbstractProduct, AbstractFactory.factoryMethod\}$
 Binary relations: $INHERIT = \{(FactoryA, AbstractFactory), (FactoryB, AbstractFactory), (ProductA, AbstractProduct), (ProductB, AbstractProduct)\}$
 $SIGNATUREOF = \{(factoryMethod(), AbstractFactory.factoryMethod), (factoryMethod(), FactoryA.factoryMethod), (factoryMethod(), FactoryB.factoryMethod)\}$
 $MEMBER = \{(AbstractFactory.factoryMethod, AbstractFactory), (FactoryA.factoryMethod, FactoryA), (FactoryB.factoryMethod, FactoryB)\}$
 $PRODUCE = \{(FactoryA.factoryMethod, ProductA), (FactoryB.factoryMethod, ProductB)\}$

Explanation:

This will fail as the required hierarchies do not exist in the Design Model, in fact the Design Model displayed above is equal in content to what is called the Finite Structure of the given program. They do not exist because higher dimensional entities are not automatically constructed, and the user must manually create them. Although it is a point of interest that there is currently work in the subject of generating specifications from Design Models by Epameinondas Gasparis as part of his PhD thesis.

Another case for verification to fail is if the abstract class `AbstractFactory` were to not include the abstract method `AbstractFactory.factoryMethod`. In this case verification would fail because the specification has the superimposition of a signature over the hierarchy `Factories`. This indicates that all 0 dimensional entities included in that hierarchy must include a method that has the given signature, which in this case is `factoryMethod()`.

3.2 Intensional verification case study

3.2.1 Specification (Ψ)

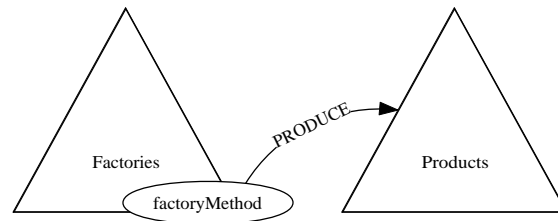


Figure 3.2: Intensional specification of the Factory Method pattern

3.2.2 Program code and Design Model (\mathcal{M})

For both of the following examples of verifying Intensional specifications the code and complete Design Model provided for the "pass" example in the above section "Extensional verification case study" should be examined.

3.2.3 Discussion:

The implementation given would pass verification given this specification if and only if the user has supplied a set of assignments between the variable declarations and the logical names of the entities stored in the Design Model. With this information an equivalent Extensional specification can be constructed which can be verified appropriately. Without assignments for every declared variable there is no way to proceed with the verification process, and it should fail before attempting to do so.

Perhaps one area of further study is to research how to automatically create a set of possible assignments given a specification and a Design Model.

3.3 Extreme inheritance case study

3.3.1 Specification (Ψ)

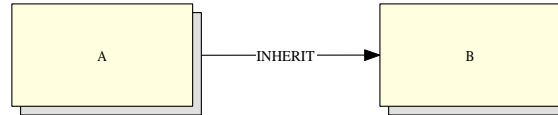


Figure 3.3: Extensional specification of an extreme inheritance case

3.3.2 Pass

Program code

The code for this example:

```
package case02pass02;
public class A1 {}
```

```
package case02pass02;
public class A2 {}
```

```
package case02pass02;
public abstract class B1 extends A1 {}
```

```
package case02pass02;
public abstract class B2 extends A2 {}
```

Design Model (\mathfrak{M})

0 Dim classes: A1, A2, B1, B2

1 Dim classes: A = {A1, A2}

B = {B1, B2}

Unary relations: *ABSTRACT* = {B1, B2}

Binary relations: *INHERIT* = {(B1, A1), (B2, A2)}

3.3.3 Fail

Program code

The code for this example:

```
package case02fail01;
public class A1 {}
```

```
package case02fail01;
public class A2 {}
```

```
package case02fail01;
public abstract class B1 {}
```

```
package case02fail01;
public abstract class B2 {}
```

Design Model (\mathcal{M})

0 Dim classes: A1, A2, B1, B2

1 Dim classes: A = {A1, A2}

B = {B1, B2}

Unary relations: *ABSTRACT* = {B1, B2}

3.3.4 Discussion

The specification above shows two 1 dimensional classes, and a single Total predicate inheritance relation between them.

The definition of the Total binary relation states that "for each non-abstract $x \in X$ there exists some $y \in Y$ such that $\mathcal{R} \rightarrow (x, y)$ "[Edea]. This is logical based on examples, such as abstract signatures cannot be included in an *INVOKE* relation for example, it should be ignored during verification. However for the *INHERIT* relation there is a problem with this definition.

Take for example the program and Design Model stated under the Pass section above. All members of the domain are abstract and yet they still inherit from the range. Logically and semantically this should pass verification, yet there are no valid entities in the domain to satisfy the definition of a Total relationship and strictly should fail.

Also examine the Fail section above where the entire domain is abstract again, but none of

the domain are in the *INHERIT* relation to the specified range in the specification. In this case according to the definition of a Total relation and all semantic reasoning, verification should fail.

Solving this is an ongoing argument between abstraction and semantic meaning, where both sides are valid and acceptable responses to the problem. On the one hand it seems logical to allow the design of the verification package to enable and disable the inclusion of abstract entities based on the relation that is being verified. However another perfectly acceptable approach would be to have the assumption that for a given predicate relation there must be at least one non-abstract entity in its domain (and range if the relation is Isomorphic) that complies to the definition of that predicate relation for it to be considered valid.

Part II

Package Development

Chapter 4

Specification

”A specification that will not fit on one page of 8.5x11 inch paper cannot be understood.”

Mark Ardis[[quo](#)]

”In preparing for battle I have always found that plans are useless, but planning is indispensable.”

Dwight D. Eisenhower (1890 - 1969)[[quo](#)]

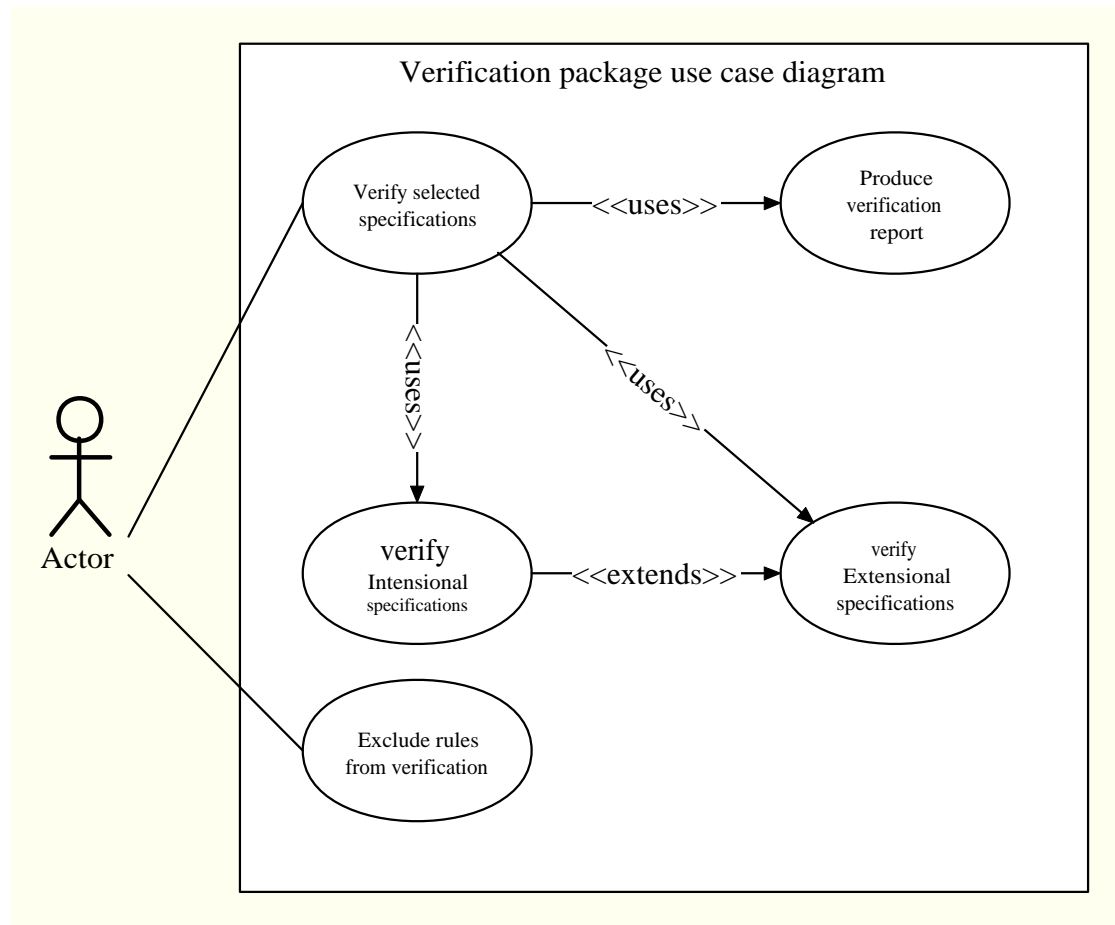


Figure 4.1: UML use case diagram for the ttp.verifier package

4.1 Functional requirements

Based on the use case given in figure 4.1, the following list of functional requirements has been produced. These requirements are based on those submitted within the Initial Proposal for CC402 previously submitted[Nic06].

4.1.1 Must have

This is a list of essential, or primary, features that the verification implementation must have by the end of the dissertation.

1. It must work with both extensional and intensional specifications
2. It must follow the mathematical equations for verification and correctness laid out in the theory section.

3. It must not verify Right and Left Exclusive rules, these are too advanced for the scope of this dissertation.
4. It must be maintainable and be able to be updated as *ClassZ* evolves.
5. It must have some method of reporting to the user passes or failures the verification process, including which rules the verification fails on and why.

4.1.2 Nice to have

This is a list of secondary features that the verification implementation could have by the end of the dissertation, dependent on time constraints.

1. Storing the verification rules in an external and easily modifiable format, such as in an XML file.
2. Ability to save the report as a file (possibly in XML or xHTML format).

4.1.3 Features for the next version

This is a list of tertiary features that the verification implementation could have once the dissertation has been completed. These requirements are not expected to be implemented, however effort will be taken to allow the verification package to be updated to include these features at some point in the future.

1. Where applicable, cross-referenced "hotlinks" to the specific sections of the code which does not verify to a specification.
2. Optional verification of sections of a specification (comparable to a superset in mathematics), effectively making a sub-specification (comparable to a subset in mathematics). For example when working on a large system it might be desirable to verify only a section of the architecture.
3. Optional disabling of certain verification rules. For example if it were desirable to simply check that the correct architecture (entities and inheritance rules) are verified to be correct, but verification of other relations (such as call or return) might be irrelevant at that time.

4.2 Non-functional requirements

1. The Verification package must reside and operate within the TTPToolkit, using the same method of inter-package communication as all other packages within the system.
2. The verification "engine" should be placed in the package `ttp.verifier`.
3. As this is a research project, there is no need to make the code as efficient as possible[TE]. However it should be as logical and maintainable as possible.

Chapter 5

Design

”To design is to communicate clearly by whatever means you can control or master.”

Milton Glaser[[quo](#)]

”A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.”

Douglas Adams (1952 - 2001)[[quo](#)]

5.1 The verification algorithm

Figures 5.1, 5.2 and 5.3 are abstracted flowchart representations of the verification algorithm. They can not show all details due to size limitations, but a lot of effort has been used to try to put as much information into these diagrams as possible while still being readable on the printed page.

As can be seen in these flowcharts, verification is split into three main processes as described in the theory section previously. These are Existence, Impositions, and Relations respectively. Although there is only one "validation" decisions shown on each chart, there will in fact be more than that. The use of one for each set of validation rules has been used here for the purposes of abstraction. It also does not show specifically how relations will be verified, which is discussed later in this chapter. The general purpose of this diagram is to show roughly how the process of validation shall be executed in respects to what iterations are required.

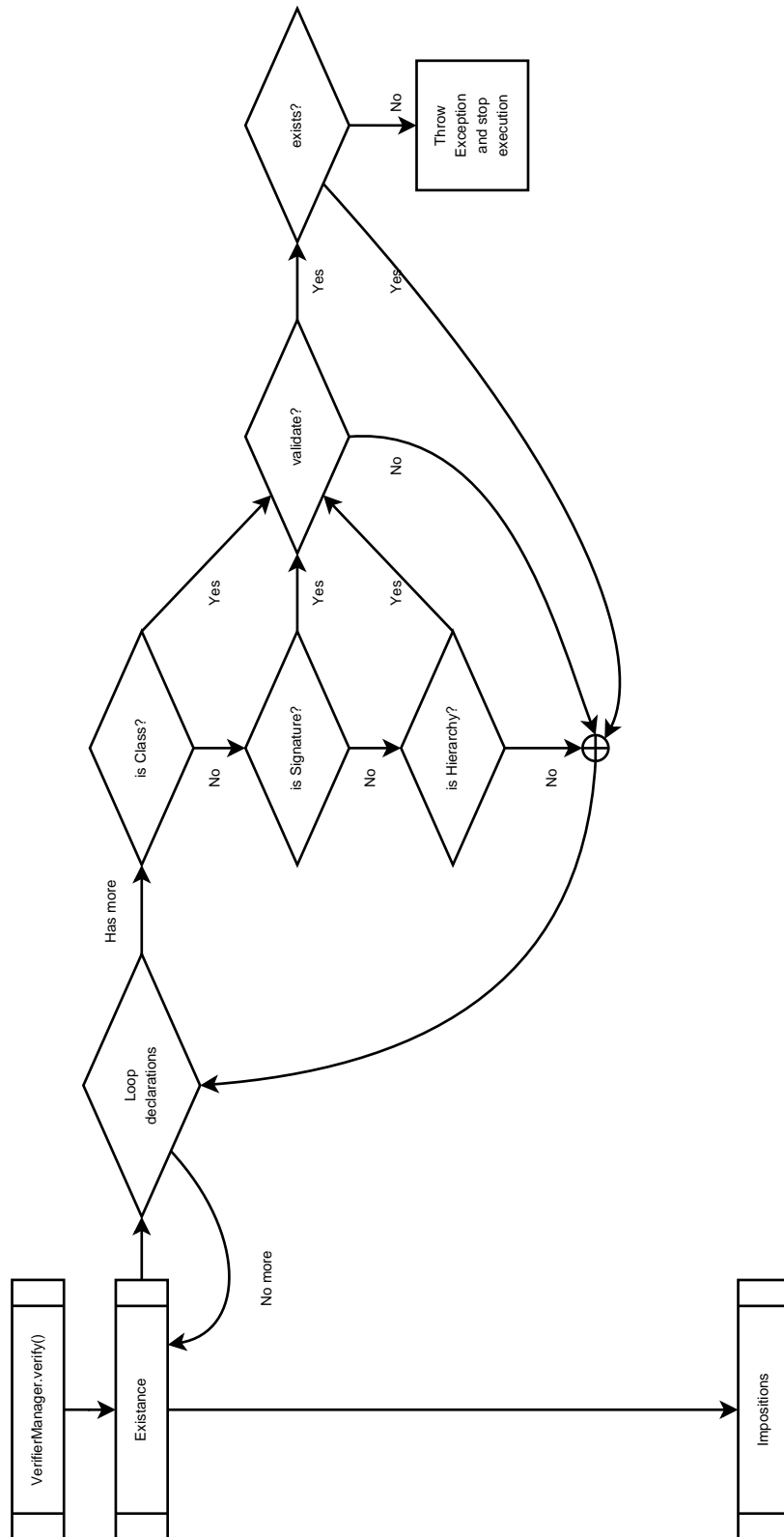


Figure 5.1: Flowchart showing an abstraction of the Existence rules in the verification algorithm

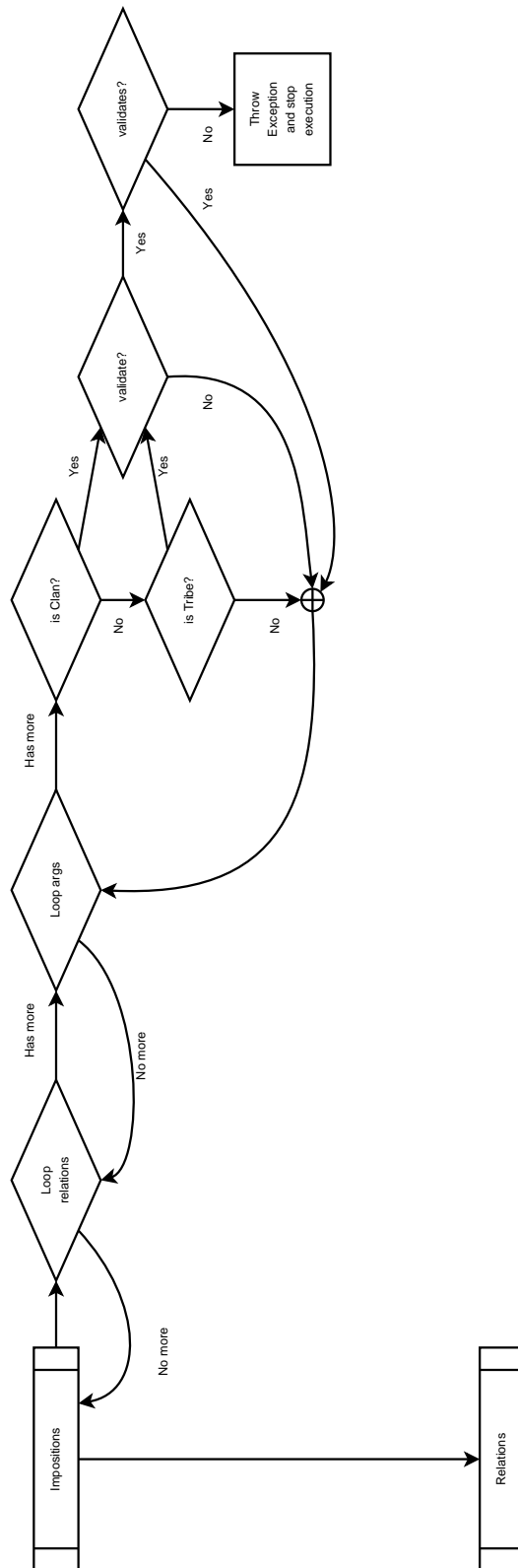


Figure 5.2: Flowchart showing an abstraction of the Imposition rules in the verification algorithm

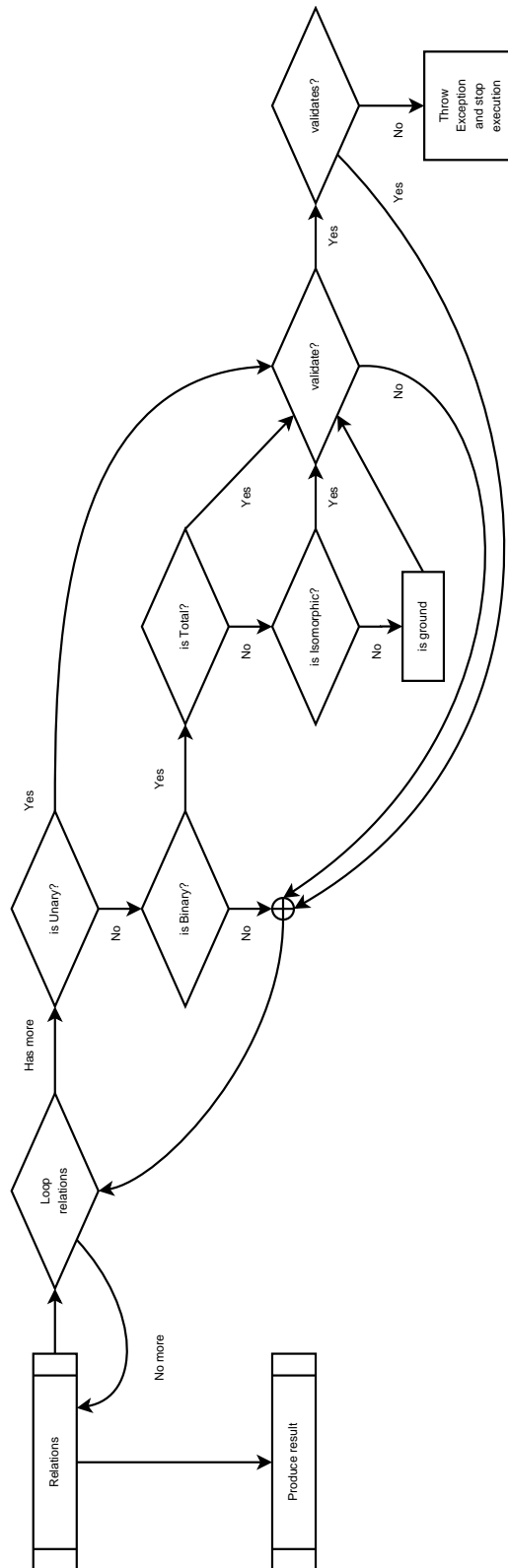


Figure 5.3: Flowchart showing an abstraction of the Relation rules in the verification algorithm

5.2 Design Policy

After examining the verification algorithm, the mathematical properties outlined in the Theory section of this document, and investigating how to implement this in Java, certain decisions had been made pertaining to how the system should be further developed. These are as follows:

1. Each method should be as simple as possible, using overloading where required.
2. Methods used to interface with, and provide logic for, the Design Model should be kept in a single `final` class, where each one is `protected` and `static`. This provides only members of the `ttp.verifier` package access to these methods, and keeps the methods within the main verification engine as simple as possible.
3. Rather than using return types to indicate if validation passes or fails, exceptions should be used. The main advantage of this is that exceptions do not need to be explicitly handled by intermediary stages. If required certain verification steps can catch them and throw their own using the caught exception as detailed information, in other words the `try...catch...` blocks can be used in a similar fashion to `if...then...` blocks. This allows for extremely long chains of execution to be carried out and escaped cleanly with minimal added execution time.

5.3 Package dependencies

The following two diagrams show the package dependencies of `ttp.verifier`. The first is an abstract block diagram for a more general representation, the second is a UML diagram but has slightly more detail, explicitly stating which classes from `ttp.gui` depend on `ttp.verifier`. For ensure clarity, the arrows point in the direction of the dependency.

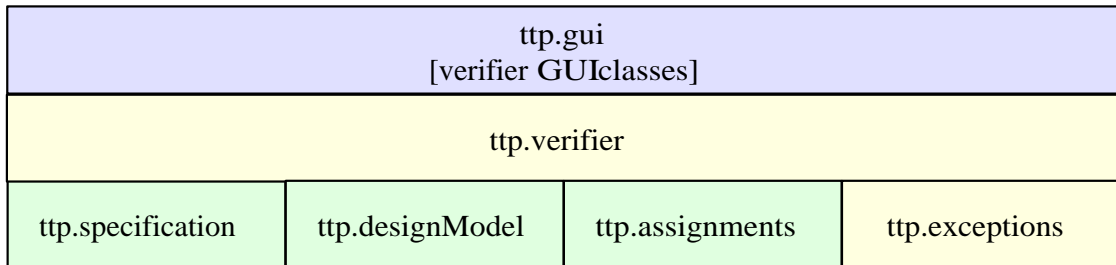


Figure 5.4: Package dependencies for the `ttp.verifier` package

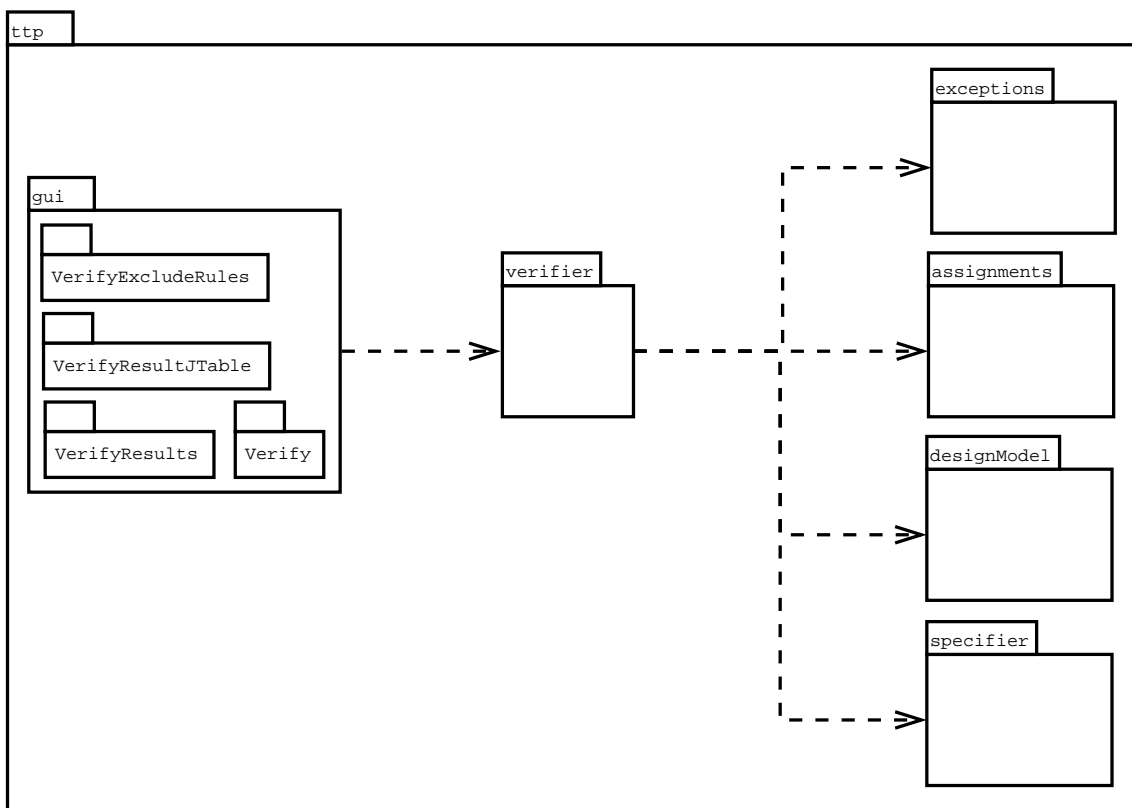


Figure 5.5: Package dependencies for the `ttp.verifier` package in UML

5.4 Package design

The entire `ttp.verifier` package can be represented using the *ClassZ* chart in figure 5.6. The main signature is entitled `verify` on the class `VerifierManager`. This fires a batch of tests on a given Extensional specification. Although it is not explicitly represented, Intensional specifications are first converted to Extensional ones, and then verified in the same way. Each verification test tries to implement the mathematical rules given under the relevant chapter seen previously, and follows the algorithm given in figures 5.1, 5.2 and 5.3.

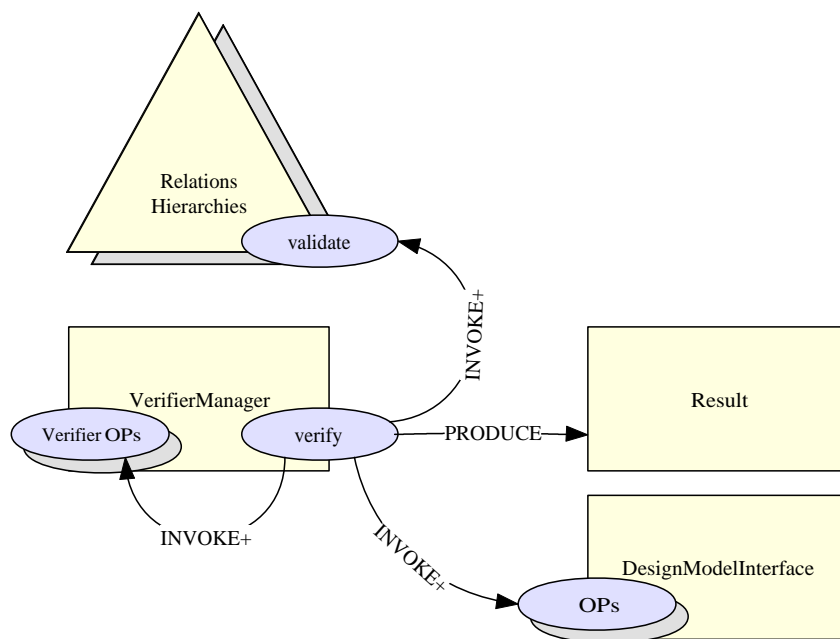


Figure 5.6: *ClassZ* diagram showing the verifier package

Verifying relations

Figure 5.7 shows a *ClassZ* diagram of the process involved in verifying relations. When validating a relation its relevant validation class is obtained by name from a collection of known relations. Once retrieved its `validate` method is invoked, which provides the logic required to process the relation for correctness as outlined in the Theory section of this report. If a relation fails to validate, and no alternatives validate either, then a relevant and explanatory exception is thrown.

Ground, Total, and Isomorphic rule checking is done in the superclass of each of these relation classes, providing a single store for the predicate/relation validation. Where as the `validate` method implemented in each of the relation classes tries to implement the correctness equations seen previously. This separation of validation logic provides a much more maintainable system, that can be updated with new relations quickly and easily.

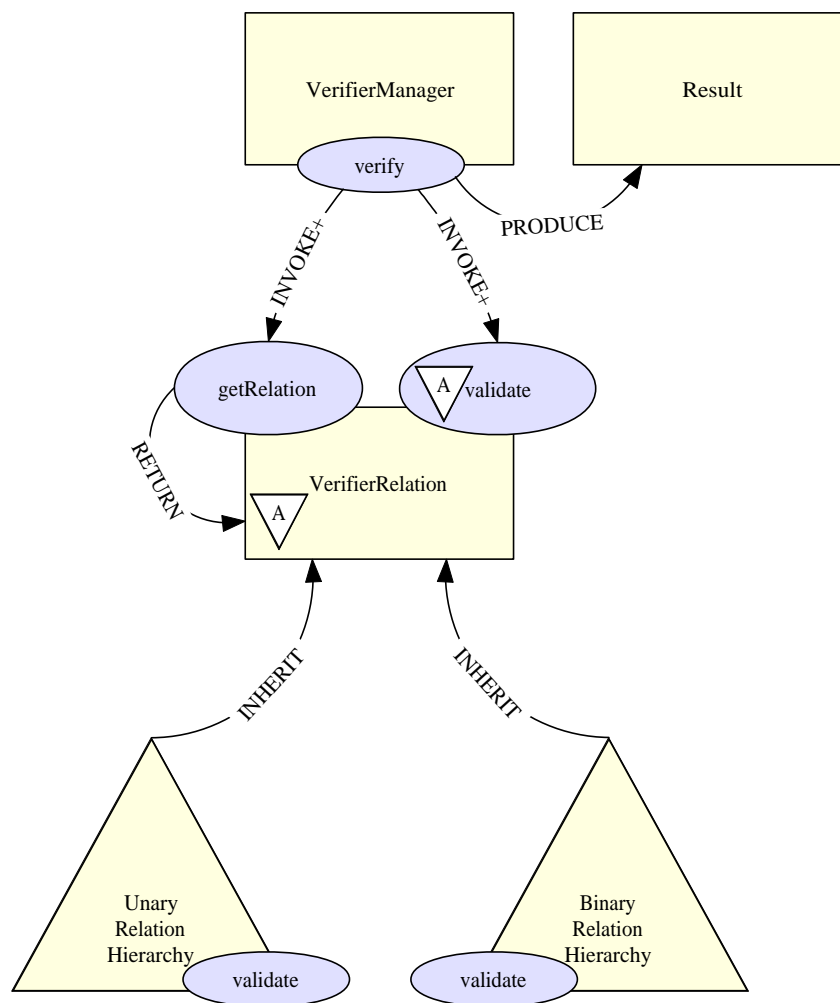


Figure 5.7: Detailed *ClassZ* diagram showing the verification of relations

5.5 Interface to the verifier package in the TTPtoolkit

To be able for the rest of the TTPToolkit to communicate with the `ttp.verifier` package, command classes have to be created which inherit from the abstract class `ttp.Command`, which follows the design pattern "Command". The following table shows the set of commands belonging to the `ttp.verifier` package.

Command:	<code>executeVerificationCommand</code>
Accessed by:	<code>executeVerification</code>
Parameters:	Extensional specifications: <code>Object[] {Specification}</code>
	Intensional specifications: <code>Object[] {Specification, Assignment[]}</code>
Returns	<code>Result</code> or null if no parameters given
Description:	Run verification on a given Extensional or Intensional specification
Command:	<code>excludeDuringVerificationCommand</code>
Accessed by:	<code>excludedDuringVerification</code>
Parameters:	<code>Rules.Rule[]</code>
Returns	<code>null</code>
Description:	Exclude the set of rules from the verification process
Command:	<code>getExcludedVerificationRulesCommand</code>
Accessed by:	<code>getExcludedVerificationRules</code>
Parameters:	<code>LinkedList<Rules.Rule></code>
Returns	<code>null</code>
Description:	Get the set of excluded verification rules
Command:	<code>isExcludedDuringVerificationCommand</code>
Accessed by:	<code>isExcludedDuringVerification</code>
Parameters:	<code>Rules.Rule[]</code>
Returns	<code>Boolean[]</code>
Description:	Check to see if a given rule is excluded during the verification process

Table 5.1: Interface commands to the `ttp.verifier` package

Chapter 6

Testing

Testing is a fundamental part of programming, along with other techniques that can help increase the confidence that a given program is correct according to its specifications and expected use. Aside from testing incrementally during development a system should always undergo sufficient testing for the purpose to which it has been designed.

6.1 Testing strategy

Due to delays, both technical and personal in nature, the amount of time scheduled for dedicated testing had to be reduced. Originally the plan outlined in the initial proposal called for test cases, code inspection, unit, stress and acceptance testing. This had to be stripped to a minimum without great loss to the dissertation and without hindrance to the TTPToolkit.

”It’s not enough that we do our best; sometimes we have to do what’s required.”

Sir Winston Churchill (1874 - 1965)[[quo](#)]

Therefore it was decided that all testing would be ignored with the exception of code inspection and test cases. The combination of these two forms of testing should yield results suitable for the current task, and still provide enough insight into the operation of the system such that major bugs and errors can be identified.

6.2 Discussion of code inspection

Upon inspection of the code the most noticeable trait is that it is rather complex. Some of which can be attributed to how the data is stored and handled in other packages, namely the Specification and Design Model packages. For example an `Entity (e)` in the Design Model has a method `getEntities()`, which intuitively should return an array of the entities which directly constitute the entity `e`. The same `getEntities()` method can then be invoked on each of the entities in this array, eventually leading to the production of a set of 0 dimensional entities. However this is not the case, as the `getEntities()` method does not return an array of actual entities, but rather placeholders that hold enough information so that each entity can be retrieved. This has often lead to `NullPointerExceptions` during development, as each of these placeholders must be converted to real entities before they are used. This complicates the code, and makes it less efficient as occasionally it is possible for the same entity to be retrieved more times than is necessary.

Also within the specification a `Relation` object is stored within a `Statement` object, along with its `Predicate` object if it has one (null if it does not). This seems to be logical according to the LePUS DTD file on which the current TTPToolkit’s specification package is based. However

it seems that it would be more logical to update the system and have the `Predicate` object stored within the `Relation` object. This is perhaps a controversial point as it would fairly radically change the structure of a given specification, and hence require changes to all packages which depend on `ttp.specification`.

Aside from these points, during the code inspection, no glaring omissions or errors came to light. However it did become apparent that the addition of the `Left` and `Right Exclusive` rules might pose a future problem. Currently there is no way to represent `Left` and `Right Exclusive` rules in an XML specification, and therefore it is assumed that these are not handled in the current `ttp.specification` package. Until such time that these are representable both in the XML specifications, and within the specification Object its self, it is impossible to predict how to modify the `ttp.verifier` package to validate such rules.

6.3 Discussion of test case results

As is well known, one method of testing does not produce all bugs/errors that a system may contain. This is certainly the case here, where code inspection has failed to put light on a very serious issue in the TTPToolkit. For reference the test cases and their results can be found in Appendix A and Appendix B respectively. The main issue is that of validating the relations *PRODUCE*, *CREATE* and *RETURN*, this is shown in the failure of test cases:

- Case 03 code 02
- Case 03 code 03
- Case 04 code 01
- Case 04 code 02 and
- Case 04 code 03

Which all resulted in the following exception:

```
Exception in thread "AWT-EventQueue-0" java.lang.OutOfMemoryError:  
Java heap space
```

Upon investigation it appears the assumption that a class must be the range of each of these relations is incorrect according to the currently implemented Design Model. At the moment it seems that the range of these relations is a method, namely the constructor of the expected class, which is logically incorrect. To solve this either the Design Model must change or the verifier must change,

but correcting this would require more time than is available, and so is left as a requirement of the next version.

This does not by its self explain why the heap space overflows however, but it does shed some light onto its cause. Upon investigation it was found to be an oversight on my part, an infinitely recursive loop was found which verifying the *PRODUCE*, *CREATE* and *RETURN* relations. Why this happens can be explained in the following example:

- Take for example a *PRODUCE* relation.
- If *PRODUCE* cannot be validated, and given the problem above it never will, the following rule comes into play:

$$PRODUCE(dom, ran) \Leftrightarrow CREATE(dom, ran) \wedge RETURN(dom, ran)$$

Which means the verifier now looks for a *CREATE* and a *RETURN* relation instead.

- Again because of the previous error, these relations cannot be found and alternatives are sought.
- Due to the rules: $PRODUCE(dom, ran) \Rightarrow CREATE(dom, ran)$
and
 $PRODUCE(dom, ran) \Rightarrow RETURN(dom, ran)$
the verifier now looks for two *PRODUCE* relations, which as already known will always fail.
- This continues recursively, eating at the memory allocated to the heap stack until it crashes.

The solution to this problem is the addition of a stopping variable, perhaps local to the verifier, or passed between methods, which will prevent this infinite recursion. The variable could be of type `int` for example, and cause an exception if it reaches a maximum number. Three should allow for validation to complete while preventing the recursive loop.

While working on these test cases it also became apparent that only transitive inherit can be expressed in a specification, due to how the package `ttp.specifier` reads in a specification. In the test case 08 (Code 01 and 02) transitive *INVOKE* had been attempted, however this was always changed to transitive *INHERIT* within the specification stored in the TTPToolkit. Obviously this will need to be changed in future revisions.

It was also seen that when a relation is marked as transitive, its name changes to incorporate the $^+$ symbol. This caused a problem in the verifier as only *INHERIT* had been created, which would handle transitivity depending on the boolean value returned by the method `isDerviedRelation()`. To overcome this another verification class was inserted to handle the *INHERIT*⁺ relations, called `InheritP`. However a much more logical solution would be to not change the name of a given relation depending on its transitive properties. Therefore *INHERIT* and *INHERIT*⁺ relations both have the same name, `INHERIT`, and are then verifiable by the same verification class. This would be logical as *ClassZ* may grow in the future to include other operators such as $^+$, and the verification system should be updated accordingly. However if each time the language changes, another set of classes must be added to the verification system, it will soon become bloated and hard to manage. If the ability to retrieve the fully qualified name of the relation, including its operator, the it would be recommended to include that as a separate method to avoid this problem.

While investigating this it also came to light that in the current implementation, ground relations cannot be transitive. In theory this is not the case, these two facts about a relation are not mutually exclusive, however this is the way it appears that it is programmed within the `ttp.specification` package. That is unless the current use of `isDerviedRelation()` is incorrect. It would be nice to see the method `isTransitive()` instead as this would clear up any ambiguity.

Chapter 7

Evaluation of the Implementation

7.1 Limitations imposed on the Verification package

The main issue imposed on the `ttp.verifier` package, other than those already mentioned in the Testing chapter, is that of how methods are handled. Theoretically a method should be an entity of any dimension in the Design Model, which is substantiated by Dr. Eden's draft book[Edea]. However in the current Design Model implementation methods are only zero dimensional. Because of this they cannot be handled in the same way as other classes/signatures are used within the verifier.

To overcome this the verifier was made to allow the ability to verify relations between sets of zero dimensional entities. However when methods are being validated using this system, the resultant message reported to the user is incomplete. At least it is incomplete according to how they were designed to appear. Failed verification messages reported to the user should be in the format:

```
FAILED
Primary reason... Original test
  Secondary reason... Sub-test 1
    Tertiary reason... Sub-test 2
```

Where the original test is the information being validated from the specification, the sub-tests are then any other information from tests which might have to be performed to validate said primary reason for failure. For example a failed Tribe validation will first report the tribe which failed

validation, and then the specific class/position that fails validation. This gives the user the exact information required such that they might correct the problem. Unfortunately because of the way in which methods are handled by the Design Model, and hence within the verifier itself, messages reporting verification failures appear in the format:

```
FAILED
Secondary reason... Sub-test 1
Tertiary reason... Sub-test 2
```

Without the primary reason for failure. Therefore although it reports that, for example, there is no *CALL* relation between method x and method y , it does not report whether or not that relation belongs to a Total or Isomorphic *CALL* relation. Without this primary reason for failure it could be very difficult to see which part of the specification is failing, and hinder the users ability to correct the problem in the most appropriate fashion.

For example a failure reporting that the relation $PRODUCE(a, b)$ does not exist. Given that information it is only logical to add the relevant *PRODUCE* relation as the user has no ability to find where that is required in the specification. However if it were reported to the user that $PRODUCE(a, b)$ should be part of an Isomorphic relationship between x and y then they might decide to make b abstract, change the relation $Isomorphic(PRODUCE, x, y)$ to $Total(PRODUCE, x, y)$, or any number of other possible modifications to allow verification to complete successfully.

Another limitation is that of access modifiers on methods in the implementation, which are not represented in the theoretical or implemented Design Model. Logically it is possible for verification to succeed where it should fail in cases where methods in the implementation are `private` for example. To outline this case the following charts in figures 7.1 and 7.2 are provided, where the latter is a reverse engineered chart from a possible implementation. This example should pass if and only if the signature `s` in figure 7.2 is not `private`, as `private` methods are not inherited by subclasses.

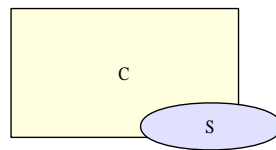


Figure 7.1: Problems with access modifiers: Specification

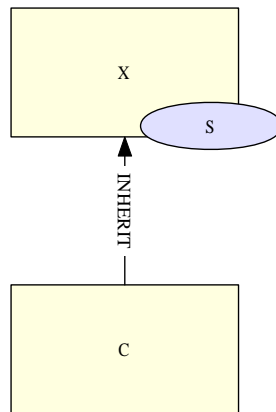


Figure 7.2: Problems with access modifiers: Implementation

In theory the Design Model should contain all information about a given class, which includes any inherited methods stored as if they belong to that class directly[Edea][Definition IX:Axioms of Object-Oriented Design]. The current Design Model however, for many reasons, does not implement this. Inherited methods are not included for each class, rather they have to be found through a process of searching the Design Model for relevant *INHERIT* relations to classes that could contain the desired signature. Therefore even if *s* is declared as `private` it will still pass validation, which is an incorrect result.

This is not true for cases such as *CALL* relations. For example take the ground relation $CALL(A, B \otimes s())$ given in a specification, if the method representing $B \otimes s()$ were declared as `private` then these classes would not compile. As they would not compile, and the TTPToolkit uses reflection to populate the Design Model, all *CALL* relations are legal and verifiable. It is believed that this is true for relations such as *PRODUCE* also, therefore inheritance is the only area where access modifiers will cause a problem.

To solve this problem the obvious solution is to make the implemented Design Model more like the theoretical version. However it has been made clear that there are issues in doing this, a discussion on which is beyond the scope of this project.

7.2 Limitations of the Verification package

There are cases where the report provided to the user may not be adequate, excluding the example above. For example the verification of an Isomorphic relationship fails, reporting that some range entity is already associated with another member in the domain. What would also be useful is to know exactly which entity in the domain has already associated to it so that the user can confirm the result. Take, for example, the case given in figure 7.3 where "Domain" and "Range" are both one dimensional classes, represented as a set of zero dimensional classes to accentuate the point. The resulting error from this would be that the Isomorphic relation is incorrect, and that class "D" is already related to another member of the domain. It would be useful, from the users perspective, to be told which class is already associated with class "D" (in this case "A") so that the modifications can be made to the program to correct this problem quickly.

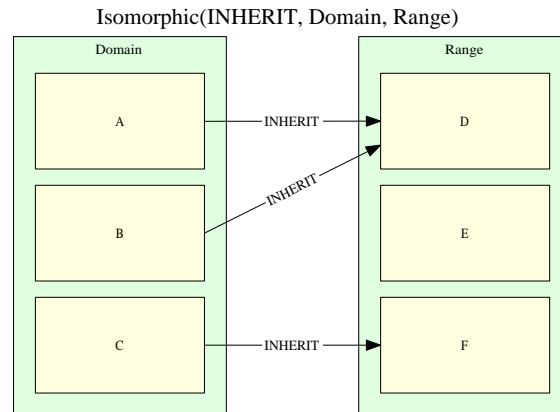


Figure 7.3: A case of failure for an Isomorphic INHERIT relation

In future revisions of the `ttp.verifier` package this can be solved by storing the found relations in a `HashMap` for example, rather than a `LinkedList` as is in the current implementation.

Finally there was a misconception during the design phase on the matter of associations. Currently the first association, and therefore the first set of assignments, are used to produce an Extensional specification from its original Intensional form. It has become apparent that this is incorrect, as logically an Intensional specification might map to multiple places in a program. Therefore each Intensional specification will create x number of Extensional specifications, where x is the number of assignments it currently has, compared to the one which is currently assumed. This is something which much be corrected for future versions of the `ttp.verifier` package.

7.3 Successes of the Verification package

Testing has produced very valuable results, not only for the `ttp.verifier` package, but also for the `TTPToolkit` in general. Testing has discovered problems in the current Design Model, Specification package, and a serious recursive error in the Verifier its self. All these errors and proposed solutions will be discussed at a later date, after the submission of this dissertation. Having noted these problems however, the successes should also be stated. Simply put the verifier does work for all Existence and Imposition rules, and some of the Relation rules. It does produce insightful error messages in all cases except where the Java heap space overflows as discussed earlier.

In terms of the original requirements, the verification package succeeds in meeting the "must have"s, and also some of the "nice to have"s. For example a feature that allows the user to enable

and disable validation rules, the choices for which are saved with the rest of the project, has been included. It validates both Intensional and Extensional specifications, and it also operated completely within the TTPToolkit without any dependencies outside the system. Therefore, in my opinion, it meets the originally defined functional and non-functional requirements.

7.4 Advancements to the Verification package

Everything in the package has been programmed using pure Java, however the problem might be more suitably solved using a logic/mathematical language such as Prolog. Systems exist to allow the inclusions of some such programming languages into Java, for example PrologCafe[BT] is one of many libraries that allow the inclusion of Java in PROLOG, and PROLOG in Java. This would theoretically allow the verification rules and the definitions of correctness to be implemented in a more intuitive way, with possibly more simplicity and certainly more extendibility/maintainability. This could be a focus of further study in either another MSc dissertation or as part of a PhD.

Alternatively the use of design patterns could be considered to achieve the same result, such as the Strategy pattern[GHJV04][p.315], and/or the Interpreter pattern[GHJV04][p.243]. The combination of these two patterns might provide a method of writing a set of validation instructions in an external file, that can be interpreted and dynamically created as a set of strategies for validation.

Chapter 8

Conclusions

As discussed in the previous chapter, the verification system works despite the problems within the implementation and those imposed on it from other parts of the TTPToolkit. What has been accomplished meets a great deal of the goals set out at the start of this dissertation. The `ttp.verifier` package has been left in such a way that the improvements that need to be made can be done fairly easily within the next revision of the TTPToolkit.

In terms of *ClassZ*, through my investigation and understanding, it currently lacks the ability to model Exceptions. As I believe a vast majority, if not all, Object Oriented (OO) languages use Exceptions for error reporting; it would be wise to consider the addition of a new relation to the *ClassZ* language. Exceptions are different to return types, and as such should be modeled differently. For example this could be named *THROWS*, and to use it in an example the *ClassZ* chart of this system could be advanced from figure 5.6 to figure 8.1.

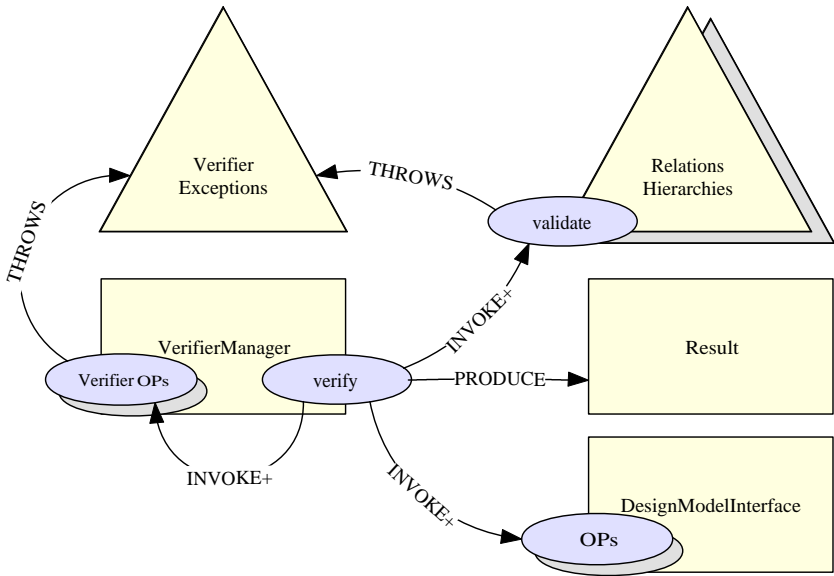


Figure 8.1: *ClassZ* diagram showing the verifier package including the *THROWS* relation

In terms of Project Management I believe I was able to keep to my schedule. There were initial delays such as the creation of the TTPToolkit V0.3 and our exams. There was also a great deal of in depth knowledge that needed to be accumulated on the subject before any development could take place. The amount of which was underestimated at first. I also had some set backs where a simple misunderstanding caused days, and in one case a week, of confusion that could have been put to better use. Despite these setbacks, and with the help of my colleagues, I managed to regain momentum and get back onto schedule. To summarize I think i have managed this project well, and in comparison to my undergraduate dissertation I have greatly improved my skills in this area.

Finally I would like to say that it has been a great honor, nay delight, to work alongside those who have stretched my understanding, and to participate in active research. The experience has taught me a great deal, and I hope to advance my knowledge in this area as I continue in my studies.

Bibliography

- [Baa02] Sara Baase. *A Gift of Fire: Social, Legal, and Ethical Issues for Computers and the Internet*. Prentice Hall, second edition, 2002. URL: <http://www-rohan.sdsu.edu/faculty/giftfire/>.
- [Bo04] Gu Bo. An analysis tool for java programs. Master's thesis, University of Essex, England, 2004. URL: <http://ttp.essex.ac.uk/literature/Bo-dissertation.doc> [cited March 2006].
- [BT] Mutsunori Banbara and Naoyuki Tamura. Prologcafe [online, cited August 2006]. URL: <http://kaminari.scitec.kobe-u.ac.jp/PrologCafe/>. A Prolog to Java translator system, with the ability to call Prolog from Java and Java from Prolog.
- [Dij72] Edsger Dijkstra. The humble programmer. *Communications of the ACM archive*, 15(10):859 – 866, October 1972. One of the earliest uses of the term 'Software Crisis'.
- [Edea] Amnon Eden. Object oriented modelling - combining mathematical logic and abstraction for visualizing the building blocks of object-oriented design. Draft copy dated 19 July 2006.
- [Edeb] Amnon Eden. Two-tier programming project website [online, cited March 2006]. URL: <http://ttp.essex.ac.uk/>.
- [EKF03] Amnon Eden, Rick Kazman, and Chris Fox. Two-tier programming. Technical report, University of Essex, England, 2003. Technical report CSM-387, Department of Computer Science, University of Essex. URL: [http://www.eden-study.org/articles/2003/ttp\(csm-387\).pdf](http://www.eden-study.org/articles/2003/ttp(csm-387).pdf) [cited March 2006].
- [Epp95] Susanna Epp. *Discrete Mathematics with Applications*. Wadsworth Group, second edition, 1995.
- [Fin] Anthony Finkelstein. London ambulance service computer aided despatch system [online, cited August 2006]. URL: <http://www.cs.ucl.ac.uk/staff/A.Finkelstein/las.html>. interesting site maintaining information on the London Ambulance system failure of 1992.
- [GHJV04] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2004.
- [Iya03] Olumide Iyaniwura. A verification tool for object-oriented programs. Master's thesis, University of Essex, England, 2003. URL: <http://ttp.essex.ac.uk/literature/Iyaniwura-dissertation.doc> [cited March 2006].

- [Jav] Sun Java. Java 1.5.0 api reference documentation [online, cited March 2006]. URL: <http://java.sun.com/j2se/1.5.0/docs/api/>. Java reference material used while programming the applications.
- [Kab] Dr. Heinz M. Kabutz. Multi-line cells in the jTable [online, cited 24/08/2006]. URL: <http://www.javaspecialists.co.za/archive/newsletter.do?issue=045>. Page that describes how the achieve multiple lines of text in a jTable.
- [Lia04a] Maple Tao Liang. Specification module for the ttp toolkit. Master's thesis, Chalmers University of Technology, Sweden, 2004. URL: <http://ttp.essex.ac.uk/literature/Liang-thesis.doc> [cited March 2006].
- [Lia04b] Maple Tao Liang. Xml2lepus package of ttp toolkit. Master's thesis, Chalmers University of Technology, Sweden, 2004. URL: <http://ttp.essex.ac.uk/literature/Liang-XML2LepusReport.doc> [cited March 2006].
- [Nic] Dr. Hugh Nicholson. Personal communication during the course of this dissertation.
- [Nic06] Jonathan Owen Hugh Nicholson. Cc402 - initial proposal. The dissertation proposal submitted as part of the CC402 course, June 2006.
- [Pol81] Wolfgang Polak. *Compiler specification and verification*. Springer-Verlag, 1981. From the 'Lecture notes in computer science' series of books, number 124.
- [quo] The quotations page [online, cited September 2006]. URL: <http://www.quotationspage.com/>. A site used to find quotes for chapter headings.
- [TE] Ray Turner and Amnon Eden. Personal communication during the course of this dissertation.

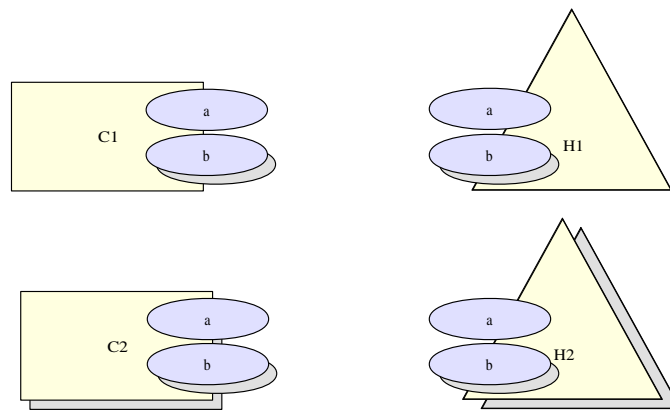
Appendices

Appendix A - List of test cases

Rather than including all the source code of each of the following cases in the printed report, they are provided on the enclosed CD in a directory called *Test Cases*. Instead Design Models are given.

Case 01 - Existence, Clans and Tribes

Specification (Ψ)

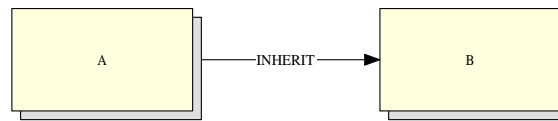


Design Model (\mathcal{M}) code 01 - PASS

0 Dim classes: C, X, Y, RootA, RootB
 0 Dim signatures: a(), b1(), b2()
 1 Dim classes: C2 = {X, Y}
 1 Dim signatures: b() = {X(), Y()}
 1 Dim hierarchies: H1 = {RootA, C2}
 Hx = {RootB, C2}
 2 Dim hierarchies: H2 = {H1, Hx}
 Unary relations: *ABSTRACT* = {RootB}
 Binary relations: *INHERIT* = {(X, RootA), (X, RootB), (Y, RootA), (Y, RootB)}
 SIGNATUREOF = {(a(), C.a), (b1(), C.b1), (b2(), C.b2),
 (a(), RootA.a), (b1(), RootA.b1), (b2(), RootA.b2),
 (a(), RootB.a), (b1(), RootB.b1), (b2(), RootB.b2)}
 MEMBER = {(C.a, C), (C.b1, C), (C.b2, C), (RootA.a, RootA),
 (RootA.b1, RootA), (RootA.b2, RootA), (RootB.a, RootB),
 (RootB.b1, RootB), (RootB.b2, RootB)}

Design Model (\mathcal{M}) code 02 - FAIL

0 Dim classes: C, X, Y, RootA, RootB
 0 Dim signatures: a(), b1(), b2()
 1 Dim classes: C2 = {X, Y}
 1 Dim signatures: b() = {X(), Y()}
 1 Dim hierarchies: H1 = {RootA, C2}
 Hx = {RootB, C2}
 2 Dim hierarchies: H2 = {H1, Hx}
 Unary relations: *ABSTRACT* = {RootB}
 Binary relations: *INHERIT* = {(X, RootA), (X, RootB), (Y, RootA), (Y, RootB)}
 SIGNATUREOF = {(a(), C.a), (b1(), C.b1), (b2(), C.b2),
 (a(), RootA.a), (b1(), RootA.b1), (b2(), RootA.b2),
 (b1(), RootB.b1), (b2(), RootB.b2)}
 MEMBER = {(C.a, C), (C.b1, C), (C.b2, C), (RootA.a, RootA),
 (RootA.b1, RootA), (RootA.b2, RootA), (RootB.b1, RootB),
 (RootB.b2, RootB)}

Case 02 - Extreme inheritance**Specification (Ψ)****Design Model (\mathcal{M}) code 01 - PASS**

0 Dim classes: $A1, A2, B1, B2$

1 Dim classes: $A = \{A1, A2\}$

$B = \{B1, B2\}$

Binary relations: $INHERIT = \{(A1, B1), (A2, B2)\}$

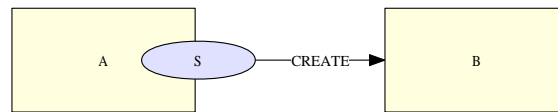
Design Model (\mathcal{M}) code 02 - FAIL

0 Dim classes: $A1, A2, B1, B2$

1 Dim classes: $A = \{A1, A2\}$

$B = \{B1, B2\}$

Unary relations: $ABSTRACT = \{A1, A2\}$

Case 03 - CREATE**Specification (Ψ)****Design Model (\mathfrak{M}) code 01 - PASS**

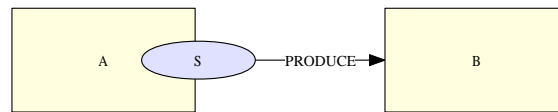
0 Dim classes: A, B
 0 Dim signatures: $s()$
 Binary relations: $CREATE = \{(A.s, B)\}$
 $SIGNATUREOF = \{(s(), A.s)\}$
 $MEMBER = \{(A.s, A)\}$

Design Model (\mathfrak{M}) code 02 - PASS

0 Dim classes: A, B
 0 Dim signatures: $s()$
 Binary relations: $PRODUCE = \{(A.s, B)\}$
 $SIGNATUREOF = \{(s(), A.s)\}$
 $MEMBER = \{(A.s, A)\}$

Design Model (\mathfrak{M}) code 03 - FAIL

0 Dim classes: A, B
 0 Dim signatures: $s()$
 Binary relations: $SIGNATUREOF = \{(s(), A.s)\}$
 $MEMBER = \{(A.s, A)\}$

Case 04 - PRODUCE**Specification (Ψ)****Design Model (\mathfrak{M}) code 01 - PASS**

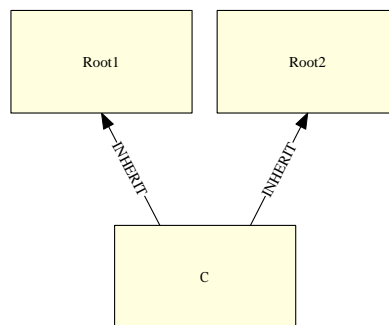
0 Dim classes: A, B
 0 Dim signatures: $s()$
 Binary relations: $PRODUCE = \{(A.s, B)\}$
 $SIGNATUREOF = \{(s(), A.s)\}$
 $MEMBER = \{(A.s, A)\}$

Design Model (\mathfrak{M}) code 02 - PASS

0 Dim classes: A, B
 0 Dim signatures: $s()$
 Binary relations: $CREATE = \{(A.s, B)\}$
 $RETURN = \{(A.s, B)\}$
 $SIGNATUREOF = \{(s(), A.s)\}$
 $MEMBER = \{(A.s, A)\}$

Design Model (\mathfrak{M}) code 03 - FAIL

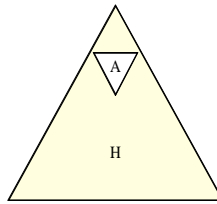
0 Dim classes: A, B
 0 Dim signatures: $s()$
 Binary relations: $SIGNATUREOF = \{(s(), A.s)\}$
 $MEMBER = \{(A.s, A)\}$

Case 05 - Multiple *INHERIT* relations**Specification (Ψ)****Design Model (\mathfrak{M}) code 01 - PASS**

0 Dim classes: $C, \text{Root1}, \text{Root2}$
Unary relations: $ABSTRACT = \{\text{Root2}\}$
Binary relations: $INHERIT = \{(C, \text{Root1}), (C, \text{Root2})\}$

Design Model (\mathfrak{M}) code 02 - FAIL

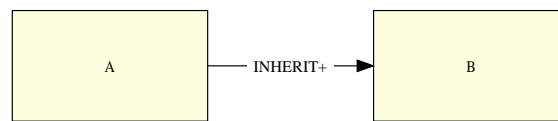
0 Dim classes: $C, \text{Root1}, \text{Root2}$
Unary relations: $ABSTRACT = \{\text{Root2}\}$

Case 06 - Abstract hierarchy**Specification (Ψ)****Design Model (\mathfrak{M}) code 01 - PASS**

- 0 Dim classes: A, B, Root
- 1 Dim classes: $C = \{A, B\}$
- 1 Dim hierarchies: $H1 = \{Root, C\}$
- Unary relations: $ABSTRACT = \{Root, A, B\}$
- Binary relations: $INHERIT = \{(A, Root), (B, Root)\}$

Design Model (\mathfrak{M}) code 02 - FAIL

- 0 Dim classes: A, B, Root
- 1 Dim classes: $C = \{A, B\}$
- 1 Dim hierarchies: $H1 = \{Root, C\}$
- Unary relations: $ABSTRACT = \{Root, A\}$
- Binary relations: $INHERIT = \{(A, Root), (B, Root)\}$

Case 07 - *INHERIT+***Specification (Ψ)****Design Model (\mathcal{M}) code 01 - PASS**

0 Dim classes: A, B

Binary relations: $INHERIT = \{(A, B)\}$

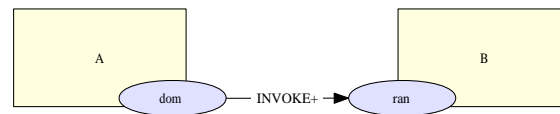
Design Model (\mathcal{M}) code 02 - PASS

0 Dim classes: A, B, X

Binary relations: $INHERIT = \{(A, X), (X, B)\}$

Design Model (\mathcal{M}) code 03 - FAIL

0 Dim classes: A, B

Case 08 - INVOKE+**Specification (Ψ)****Design Model (\mathfrak{M}) code 01 - PASS**

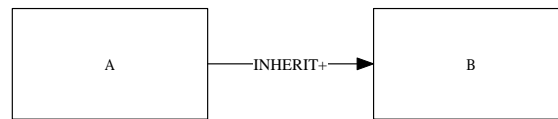
0 Dim classes: A, B
 0 Dim signatures: $\text{dom}()$, $\text{ran}()$
 Binary relations: $INVOKE = \{(A.\text{dom}, B.\text{ran})\}$
 $SIGNATUREOF = \{(\text{dom}(), A.\text{dom}), (\text{ran}(), B.\text{ran})\}$
 $MEMBER = \{(A.\text{dom}, A), (B.\text{ran}, B)\}$

Design Model (\mathfrak{M}) code 02 - PASS

0 Dim classes: A, B, X
 0 Dim signatures: $\text{dom}()$, $\text{ran}()$, $\text{sig}()$
 Binary relations: $INVOKE = \{(A.\text{dom}, X.\text{sig}), (X.\text{sig}, B.\text{ran})\}$
 $SIGNATUREOF = \{(\text{dom}(), A.\text{dom}), (\text{ran}(), B.\text{ran}), (\text{sig}(), X.\text{sig})\}$
 $MEMBER = \{(A.\text{dom}, A), (B.\text{ran}, B), (X.\text{sig}, X)\}$

Design Model (\mathfrak{M}) code 03 - FAIL

0 Dim classes: A, B
 0 Dim signatures: $\text{dom}()$, $\text{ran}()$
 Binary relations: $SIGNATUREOF = \{(\text{dom}(), A.\text{dom}), (\text{ran}(), B.\text{ran})\}$
 $MEMBER = \{(A.\text{dom}, A), (B.\text{ran}, B)\}$

Case 09 - Intensional *INHERIT*₊**Specification (Ψ)****Design Model (\mathcal{M}) code 01 - PASS**

0 Dim classes: A, B
 0 Dim signatures: $\text{dom}()$, $\text{ran}()$
 Binary relations: $INVOKE = \{(A.\text{dom}, B.\text{ran})\}$
 $SIGNATUREOF = \{(\text{dom}(), A.\text{dom}), (\text{ran}(), B.\text{ran})\}$
 $MEMBER = \{(A.\text{dom}, A), (B.\text{ran}, B)\}$

Design Model (\mathcal{M}) code 02 - PASS

0 Dim classes: A, B, X
 0 Dim signatures: $\text{dom}()$, $\text{ran}()$, $\text{sig}()$
 Binary relations: $INVOKE = \{(A.\text{dom}, X.\text{sig}), (X.\text{sig}, B.\text{ran})\}$
 $SIGNATUREOF = \{(\text{dom}(), A.\text{dom}), (\text{ran}(), B.\text{ran}),$
 $(\text{sig}(), X.\text{sig})\}$
 $MEMBER = \{(A.\text{dom}, A), (B.\text{ran}, B), (X.\text{sig}, X)\}$

Design Model (\mathcal{M}) code 03 - FAIL

0 Dim classes: A, B
 0 Dim signatures: $\text{dom}()$, $\text{ran}()$
 Binary relations: $SIGNATUREOF = \{(\text{dom}(), A.\text{dom}), (\text{ran}(), B.\text{ran})\}$
 $MEMBER = \{(A.\text{dom}, A), (B.\text{ran}, B)\}$

Appendix B - Results of listed Test Cases

Test case: 01
 Code: 01
 Expected result: Passed
 Verification result:

Passed

Test case: 01
 Code: 02
 Expected result: Failed
 Verification result:

FAILED

Required super imposition does not exist between:

Class:

Name: RootB
 Type: CLASS
 Dimension: 0

Signature:

Name: a
 Type: SIGNATURE
 Dimension: 0

Test case: 02
 Code: 01
 Expected result: Passed
 Verification result:

Passed

Test case: 02
 Code: 02
 Expected result: Failed
 Verification result:

FAILED

Violation of Total INHERIT relation (or suitable alternatives)
 between domain and range:

Domain:

Name: A
 Type: CLASS
 Dimension: 1

Range:

Name: B
 Type: CLASS
 Dimension: 1

Details: A member of the domain does not have a relation to an
 entity in the specified range:

Specific entity:

Name: A1
 Type: CLASS
 Dimension: 0

Test case: 03
Code: 01
Expected result: Passed
Verification result:

Passed

Test case: 03
Code: 02
Expected result: Passed
Verification result:

Exception in thread "AWT-EventQueue-0" java.lang.OutOfMemoryError:
Java heap space

Test case: 03
Code: 03
Expected result: Failed
Verification result:

Exception in thread "AWT-EventQueue-0" java.lang.OutOfMemoryError:
Java heap space

Test case: 04
Code: 01
Expected result: Passed
Verification result:

Exception in thread "AWT-EventQueue-0" java.lang.OutOfMemoryError:
Java heap space

Test case: 04
Code: 02
Expected result: Passed
Verification result:

Exception in thread "AWT-EventQueue-0" java.lang.OutOfMemoryError:
Java heap space

Test case: 04
Code: 03
Expected result: Failed
Verification result:

Exception in thread "AWT-EventQueue-0" java.lang.OutOfMemoryError:
Java heap space

Test case: 05
Code: 01
Expected result: Passed
Verification result:

Passed

Test case: 05
Code: 02
Expected result: Failed
Verification result:

FAILED

Required INHERIT relation (or suitable alternatives)
was not found between:

Domain:

Name: C
Type: CLASS
Dimension: 0

Range:

Name: Root1
Type: CLASS
Dimension: 0

Details:

There is no ground relation between domain and range

Test case: 06
Code: 01
Expected result: Passed
Verification result:

Passed

Test case: 06
Code: 02
Expected result: Failed
Verification result:

FAILED

Required Unary relation ABSTRACT (or suitable alternatives)
not found on:

Name: B
Type: CLASS
Dimension: 0

Test case: 07
Code: 01
Expected result: Passed
Verification result:

Passed

Test case: 07
Code: 02
Expected result: Passed
Verification result:

Passed

Test case: 07
 Code: 03
 Expected result: Failed
 Verification result:

FAILED

Required INHERIT+ relation (or suitable alternatives)
 was not found between:

Domain:

Name: A
 Type: CLASS
 Dimension: 0

Range:

Name: B
 Type: CLASS
 Dimension: 0

Details: There is no ground relation between domain and range

Test case: 08
 Code: 01
 Expected result: Passed Inability to specify program, test skipped
 Verification result:

Test case: 08
 Code: 02
 Expected result: Passed Inability to specify program, test skipped
 Verification result:

Test case: 08
 Code: 03
 Expected result: Failed Inability to specify program, test skipped
 Verification result:

Test case: 09
 Code: 01
 Expected result: Passed
 Verification result:

Passed

Test case: 09
 Code: 02
 Expected result: Passed
 Verification result:

Passed

Test case: 09
Code: 03
Expected result: Failed
Verification result:

FAILED

Required INHERIT+ relation (or suitable alternatives)
was not found between:

Domain:

Name: A
Type: CLASS
Dimension: 0

Range:

Name: B
Type: CLASS
Dimension: 0

Details: There is no ground relation between domain and range

Appendix C - Proposed *ClassZ* DTD file

Please note that this DTD does not allow Left and Right Exclusive rules to be included in the specification. This is because it is an updated version of the previous DTD designed for LePUS. Also my knowledge of Left and Right Exclusive rules are currently only basic, and more investigation will be required to be able to update the DTD so that they can be supported. This is however a good start for a new DTD file for the TTPToolit.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT specification (description?, declarations, statements)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT declarations (declaration)+>
5 <!ELEMENT declaration EMPTY>
<!ELEMENT statements (statement)*>
<!ELEMENT statement (superImposition|unaryRelation|binaryRelation)>
<!ELEMENT unaryRelation (argument)>
<!ELEMENT binaryRelation (argument, argument)>
10 <!ELEMENT argument (#IDREF|superImposition)>
<!ELEMENT superImposition (signature, class)>
<!ELEMENT signature (#IDREF)>
<!ELEMENT class (#IDREF)>
<!ATTLIST specification
15   title CDATA #REQUIRED
   >
<!ATTLIST declaration
   isConstant (TRUE|FALSE) #REQUIRED
   name ID #REQUIRED
20   dimension CDATA #REQUIRED
   type (CLASS|SIGNATURE|HIERARCHY) #REQUIRED
   >
<!ATTLIST unaryRelation
25   name (ABSTRACT) #REQUIRED
   >
<!ATTLIST binaryRelation
   predicate (TOTAL|ISOMORPHIC) #IMPLIED
   transitive (TRUE|FALSE) #REQUIRED
30   name (CALL|INVOKE|INHERIT|MEMBER|RETURN|CREATE|PRODUCE) #REQUIRED
   >

```

Appendix D - Possible bugs found in the TTPtoolkit V0.3

The following is a list of bugs found while working with the TTPToolkit V0.3 that are not discussed in other sections of the report.

Bug in: `ttp.Command`

Description: `ttp.Command` can throw an inappropriate `UnknownCommandException` when an `Exception` is thrown from the method it calls, which can lead to increased debug time attempting to solve the problem.

Date found: 09/08/06

Status: Not fixed

Bug in: `ttp.Specifier.XML2classz`

Description: Due to a combination of the currently used DTD file being designed for LePUS rather than `ClassZ` and the system not performing any semantic verification of a specification before it is loaded; invalid specifications can be entered into the system. For example the specification might contain statements about nonexistent declarations. This in turn will cause the verification process to fail with an un-handled `NullPointerException`. Possible solutions are: either verify XML specifications to at least an updated DTD file (if not a schema) before adding them to a project, or catch the `NullPointerException` and assume that it can be attributed to this.

Date found: 11/08/06

Status: Not fixed

Bug in: `ttp.designModel`

Description: Currently in the TTPtoolkit a 1 dimensional Hierarchy must be made from a 0 dimensional class, the root class, and a 1 dimensional class. To my understanding a 1 dimensional Hierarchy can consist of two 0 dimensional classes as long as the *INHERIT* relation exists between them. When testing this lead to creating a 1 dimensional class (called *C* for example) which grouped a single 0 dimensional class (*A*). When attempting to get all entities which belong to *C*, a `null` value is returned rather than the class *A*. However this works for any 1 dimensional class which groups more than one 0 dimensional classes.

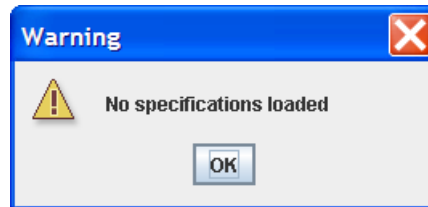
Date found: 24/08/06

Status: Not fixed

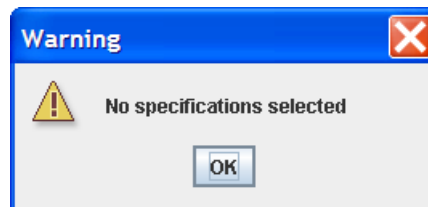
Appendix E - Graphical User Interface

Warnings

When attempting to start the verification GUI (Verify > Execute verification) without having first loaded a specification:



When attempting to start the verification process without having first selected one or more of the loaded specifications.



Exclusion of rules

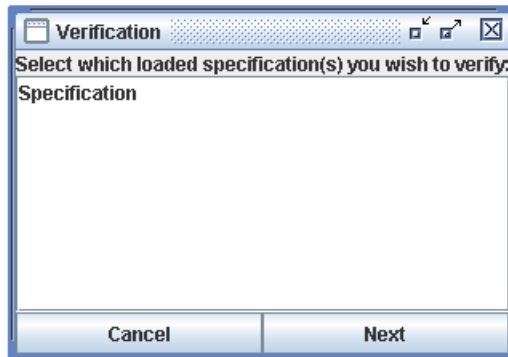
Exclusion of verification rules (Verify > Exclude rules...):

Rule:	Description:	Excluded:
classExistence	Ensure each specified CLASS exists	<input type="checkbox"/>
signatureExistence	Ensure each specified SIGNATURE exists	<input type="checkbox"/>
hierarchyExistence	Ensure each specified HIERARCHY exists	<input type="checkbox"/>
clans	Ensure each specified CLAN is correctly super imposed	<input type="checkbox"/>
tribes	Ensure each specified TRIBE is correctly super imposed	<input type="checkbox"/>
unaryRelations	Ensure specified Unary relations exist on the correct entities	<input type="checkbox"/>
binaryRelations	Ensure specified Binary relations exist correctly between entities, by disabling this you also disable Total and Isomorphic checking	<input type="checkbox"/>
totalPredicate	Ensure specified Binary relations with the predicate TOTAL exist between entities	<input type="checkbox"/>
isomorphicPredicate	Ensure specified Binary relations with the predicate ISOMORPHIC exist between entities	<input type="checkbox"/>

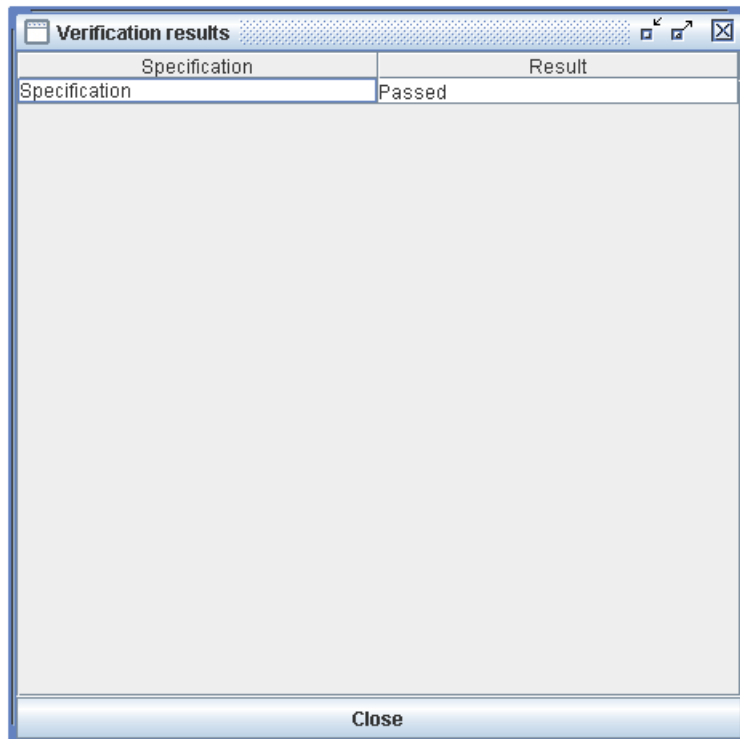
Ok

Verification process

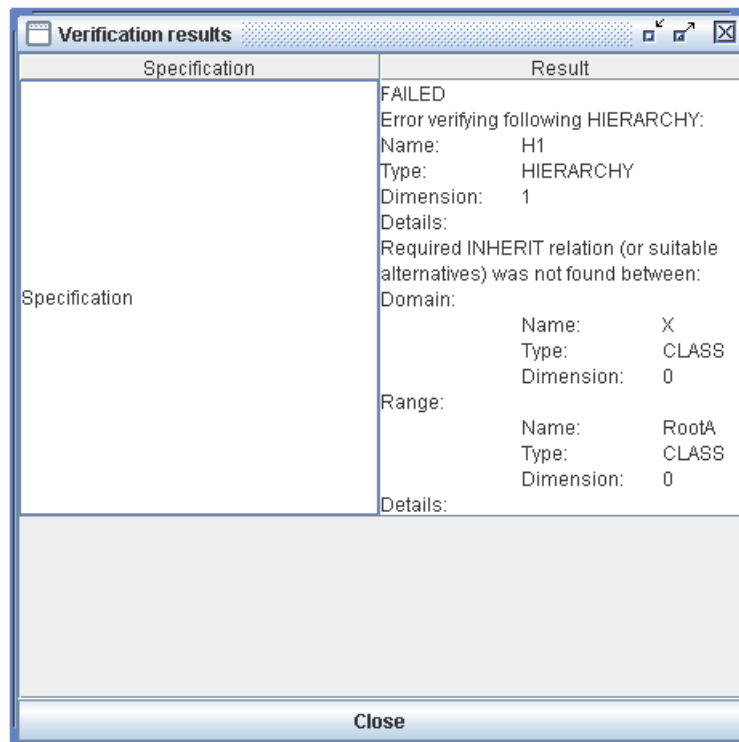
The main verification screen, where specifications are selected to be verified against the current design model:



An example of a verification pass:



An example of a verification fail:



Appendix F - CD Contents (including source code)

CD		Directory	06-Sep-2006
Abstract.doc	13445Kb	File	12-Sep-2006
Presentation.ppt	467456Kb	File	04-Sep-2006
ReadMe.rtf	1177Kb	File	12-Sep-2006
TTPToolkitV0.3		Directory	12-Sep-2006
Test Cases		Directory	06-Sep-2006
Database Backup.sql	163241Kb	File	06-Sep-2006
case01		Directory	02-Sep-2006
case02		Directory	03-Sep-2006
case03		Directory	02-Sep-2006
case04		Directory	02-Sep-2006
case05		Directory	02-Sep-2006
case06		Directory	02-Sep-2006
case07		Directory	03-Sep-2006
case08		Directory	03-Sep-2006
case09		Directory	05-Sep-2006
compile	202Kb	File	01-Sep-2006
compilesrc	93Kb	File	01-Sep-2006
formatsrc	372Kb	File	31-Aug-2006
cc401project-johnic-0531179.pdf	493039Kb	File	12-Sep-2006

Note: The file size of the report is not correct according to this image. This is due to the recursive procedure of adding the image into the report, and creating a new image accordingly. Unfortunately this is unavoidable.

For the interests of reducing ambiguity, the following source code files are my own, and all others were created by other authors. Rather than listing every class, in two cases the package is listed, indicating that every class within the package is of my own creation.

- `ttp.verifier.*`
- `ttp.exceptions.*`
- `ttp.gui.Verify`
- `ttp.gui.VerifyExcludeRules`
- `ttp.gui.VerifyResultJTable`
- `ttp.gui.VerifyResults`

No comments beyond those during development are included. This is primarily due to a lack of time, but also due to the fact that this is a research project, one which I will continue to contribute to in the future.