INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Development of a bytecode for Hewlett-Packard's "Flipper" management system including its integration through a bytecode generator and loader.

Christian Schurr

INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Diplomarbeit

Development of a bytecode for Hewlett-Packard's "Flipper" management system including its integration through a bytecode generator and loader.

Bearbeiter: Christian Schurr

Aufgabensteller: Prof. Dr. H.-G. Hegering

Betreuer: Maria-Athina Mountzia (TUM)
Jean-Jacques Moreau (HPLB)

Abgabetermin: 15. August 1997

<u>Ehrenwörtliche Erklärung</u>

Ich versichere, daß ich diese Diplomarbeit selbständig verfaßt und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 14. August 1997

…………………………….
Unterschrift des Verfassers

Abstract

This document is mainly the result of the author’s work at the HP Laboratories in Bristol in the United Kingdom. The HP Laboratories can be looked at as an intermediate stage between the universities and the business world. The author had the opportunity to spend half a year within one of the laboratory’s projects about systems management. It was intended from the beginning that the work at the laboratories was conducted by the stipulations of a German master’s thesis.

The thesis touches several subjects like distributed management, agent technology, declarative languages and of course bytecodes. To induce a certain comprehension of these, the thesis includes overviews of them as well as examples. The results of the working into the conditions of the project are also presented - at least as far as they are important for the understanding of the development of the bytecode.

This development is the main goal of the thesis and therefore described precisely. The bytecode’s integration into the management application used and designed by the project at the HP Laboratories has been carried out through the implementation of a generator and loader for the bytecode by the author, too. It is also described in this document.

1. General Introduction

This thesis has - as a work at the chair of the Technical University of Munich for network and systems management - a background with growing importance in the world of computer science.
Since computers became available, more and more companies in industry and commerce moved towards computerisation. At the beginning computers were used for only few bookkeeping, simple control and word-processing tasks, but their applications grew with their capabilities. To reach an acceptable degree of interoperability and data exchange among the computers, networks were introduced. Additionally, they offered access to valuable resources like expensive printers or special storage devices to all computer users within such a network.
Over the years, prices for computers and network components drop drastically. They became available for mid-class and small companies. At the same time the big and huge ones could afford to extend their computer networks to such an extent that finally brought the problems we have today.
Nowadays management solutions for those networks exceed their limits. The managers are confronted with huge masses of information from their management systems at a level of abstraction that is too low. While long lists of problems are well documented and even their origins are understood to a good extent, computer scientists are still searching for the Holy Grail of network and systems management.
Newer interesting concepts like the so called ‘distributed management’ and ‘agent technology’ promise to assist in some of the basic problems of management. The adaptation of known techniques from different areas of computer science tries to add new aspects and possibilities that have already proven success within these different contexts.
Hewlett Packard, for example, tries in a project at the HP-Laboratories to combine the capabilities of declarative languages to describe things in an understandable way with known ways of network and systems management and in particular also with distributed management.
This thesis has its main origin mostly from the work within this project. It is the subject of the following section, to explain its theme to the reader.

2. The Subject of this Thesis

As written on the documents envelop, the subject of this thesis is:

> *Development of a bytecode for Hewlett-Packard's "Flipper" management system including its integration through a bytecode generator and loader.*

While this title is short and gives a brief idea of the work within this thesis, it also implies various broad fields of research.
The first field is network and systems management. This justifies the fact that the thesis is embedded into the previously named chair at the Technical University of Munich, which specialises itself on this task within the computer science department; On the problems mentioned in section 1 and on possible solutions.
The same background is given for the research department at Hewlett Packard where the main work for this thesis has taken place. Their management system "Flipper" and some of the work of the Flipper project regarding distributed management eventually gives the framework for those parts of this thesis, where its general thoughts and investigations are transferred to a practical example.
A second field of research comes from the special task of management to which the work is applied. A bytecode enables HP's Flipper management system to transfer and to store its models - which are used to describe the managed environment - more efficiently than this can be achieved with the currently used plain textual form. To be able to develop such a bytecode, the subject of optimised computer language representation has to be explored. Taking into account the requirements of Flipper, this is partly specialised to the research field of representing declarative languages, because Flipper's modelling language is partly declarative.
A further field of work does not depend on the subject itself, but is part of any thesis in a comparable context. Of course the writer of a thesis has not only to work himself into the theoretical subjects, but also into the concrete project, which in this case is Flipper and its modelling language. The long way of getting into this is not directly of interest to the reader, but the understanding of the main points is. To make the transfer of the general investigations of the thesis to the Flipper project understandable, the mentioned main points are also described within this paper.

3. Structure of this Document

The table of contents already gives an overview of where to find the different chapters of this thesis and their sections. This section offers an insight into the contents of the chapters and section, to enable the reader to selectively read parts of interest.
Note that the second, more detailed table of contents can also be used as a guide into the contents of the thesis and helps to find already read passages again.

<u>Chapter II - Distributed Management</u>

Right after the general introduction into the thesis, chapter II gives an outline of the main context this thesis is embedded in, the distributed network and systems management. This subject is of course also closely linked to the area of research of the chair for which this thesis is written.
Section 1 starts with general management issues and moves towards distributed management.
Section 2 and 3 direct the focus to what is finally a direct concern of the subject of the thesis. After a general introduction into the world of agents and more specifically about intelligent agents, issues of the exchange of functionality between them are presented.

<u>Chapter III - Flipper Management Application</u>

Chapter III transfers the main points of chapter II to the concrete project in which the thesis was worked on.
To be able to do this transfer, *section 1* explains the main principles of the Flipper management application including modelling concepts.
Section 2 finally analyses the issues from chapter II for Flipper and the FML.
Section 3 adds a security aspect, which origins in special properties of Flipper.

<u>Chapter IV - Bytecodes</u>

The subject of the thesis includes not just management aspects. As finally a bytecode has to be developed, this chapter also concern itself with this - to a certain extent independent - topic.
Section 1 introduces bytecodes in general and the different kinds of bytecodes that exist.
Section 2 analyses two existing bytecodes: The internal bytecode of the Gofer functional language interpreter and the already widespread Java bytecode. It is written with regard to issues interesting for the development of the Flipper bytecode, but it may as well be read on its own.

Chapter V - Development of a Bytecode for Flipper

The biggest part of the thesis is occupied by chapter V, which eventually describes the development of a bytecode for Flipper. It is based on the demands found in the previous chapters and of course built into the framework given by the Flipper project.
First, *section 1* sums up the requirements on the bytecode.
Section 2 draws conclusions from these requirements by making several decisions concerning the bytecode as a whole.
Finally, *section 3* presents and explains the precise structure of the developed bytecode.

Chapter VI - Implementation

The last chapter describes the implementation and integration of a bytecode generator and loader into the Flipper-system.
While *section 1* gives an overview, *section 2* explains the different parts of the implementation more in detail.

Appendix

Appendix A gives some detailed information about the Flipper Modelling Languages, which are considered as HP-confidential. This appendix is not part of the publicly available version of the thesis.
Appendix B gathers all definitions of the bytecode grammar distributed all over chapter V.
Appendix C simply lists the files created during the implementation and added to Flipper. It is HP confidential as well.

CHAPTER II - DISTRIBUTED MANAGEMENT

This second chapter of the thesis describes the surroundings of the work with regard to the purpose of the system the work of the thesis is embedded in.
The analyses are independent of the specific aim to later design a bytecode for one of HP's management applications, but still the reader should always keep this is in mind. Therefore it is not the intention of this section to give a comprehensive introduction into distributed management, but to give information about this subject that influences the development of the bytecode plus some background.

1. Basics

1.1 General Introduction

Nowadays Network Management

Today's network and systems management world is dominated by centralised management solutions. The later are characterised by a comprehensive management system, called manager, that is located at a certain place and by simpler entities sitting at the managed objects, where they collect data for the manager and fulfil its orders. These entities are often called agents.
In this way, the duties are already distributed in some sense, but one must not be taken in deceive by this, because the intelligence and functionality needed for the management is totally concentrated at the manager. The only actions usually triggered by the agents themselves are notifications, used to inform the manager about a certain change of state. This is for example part of the agents' tasks in OSI management and also, but to a less extent, in the Internet management. The interested reader can find more about general management issues as well as OSI and Internet management in [HEAB94].
Resulting from this relationship between the manager and the agents is a possibly very large amount of data that has to be transported to the former. As an agent can and does not decide about the importance of portions of information, it simply sends them of. Eventually, a possibly large part of the bandwidth of the transport channels may be 'wasted' by data, carrying only little information. This is generally an even bigger problem when comparably slow links are involved.
Additionally, the manager may become overtaxed in large networks with many objects that want to be managed. Some management architectures, like OSI management, therefore also take into account the hierarchical layering of managers, where each

layer plays a kind of agent role for the next one. Still, this did not prove success in the practical world of large network environments, because the scaleability is not sufficient.
As a further disadvantage, the architecture is not very fail-safe. Wherever unreliable links must be used to transport the management data and instructions, the management itself gets unreliable. Especially parts of a network that are spatially very much separated may easily get out of control, when problems with the connections to that part occur. Additionally, the manager itself is a dangerous single point of failure. Problems with this entity disable the whole management.

Distributed Management as an Alternative

Newer investigations resulted in the wish to move away from this approach towards decentralised, distributed management. Its main idea is to disperse the managers' knowledge and capabilities, which are taken over by management applications located closely to the objects they manage. Some kind of co-operation among these distributed managers helps to correctly perform complex tasks.
As a direct result, there is no longer the need to send unfiltered information to far away managers. Simple managers may analyse and classify problems right where they occur and send reports to possibly specialised applications, which are able to trigger further steps, but are as well allowed to intervene themselves.
For the same reason, the whole system is more robust. The separation of parts of a network, due to a brake down of an intermediate transmission system for example, must not lead to a state where that part is completely unmanaged, because parts of the distributed management system are still there and functioning.
On the other hand, a total distribution may not be wanted from a business point of view. Policies must be assigned to the management system and it must be controlled from somewhere. Often the control through humans is preferred to be centralised, to reduce costs. This has to be combined with the architecture of the management system. As additional obstacle one must recognise the complexity originating from the need to co-ordinate the management applications.
General issues about distributed management are also described in [SCHÖ95].

1.2 Kinds of Distributed Management

The described form of hierarchical structures of managers is a first step to address the existing problems. Distributed management gets rid of the functions being centralised in the managers at the top layers or in the single manager, respectively.
In one variant, there is still a manager-agent relationship, similar to the one described. But one manager is responsible for few managed objects only and has no direct supe-

rior. The total set of managed objects is divided into groups, one for each manager, in a way that manager and its managed objects are also topologically closely connected. The managers are connected by a logical peer network, which they use to collectively solve their problems, to exchange knowledge, etc. To perform certain management tasks, it might still be useful to establish temporal hierarchical structures related to that task. This is allowed and does not conflict with the principally non-hierarchical organisation framework.
Another variant is the full distributed network management, where managers and agents are regarded to be at the same level. Both are fully free to exchange information or tasks with whomever they like, assumed it helps them to fulfil their duty.

A different point of view on distributed management distinguishes the different bases for the distribution. In the paragraphs above, a management application is related to managed objects. Whenever there is a problem, the application will notice this and try to solve it, possibly with the help of others.
But instead of the view on managed objects another one on knowledge can be used as the bases for distribution. Certain manager may be specialised on specific tasks, like printers, switches or special software. A manager on a PC might, for example, be able to detect problems with the locally connected printer, but as it is too costly to have the knowledge to solve the problem on every PC, a specialised manager will eventually deal with it. A combination of both bases for distribution will usually be applied in management systems.

This example leads to the next stage: Management by delegation.
It seems to be a logical step, to move the knowledge of the specialised manager to where the problem is, to avoid the same problems as in centralised management, where lots of data would have to be transferred between the location of the problem and that of the solution. This is exactly the aim of this form of management. Information about it can be obtained from [GOYE96] and [SCHÖ96], but it will be explained more in detail in the following sections.

2. Agents

2.1 Introduction

The term ‘agent’ has already been used in the previous section in the context of network and systems management. But this is not its only use. The word is heavily overloaded in the world of computer science.
It was used very early in connection with artificial intelligence, where small software-entities or algorithms that are trained to meet certain requirements are called like this.
A newer comprehension is derived from the business world, where agents represent persons or companies for a specific task or are involved to help to do a job. Even the previously described management agents can be looked at in such a way. They somehow represent a managed object for management purposes. In newer works, this kind of agents is often related to human users where the software pendants to the real agents are called ‘personal agents’. Especially in the environment of the internet and the World Wide Web, agents are used to sort and filter emails or help to find web pages of interest.
Overviews of the world of agents are given by [NWAN96] and [GIJA97].

Several attributes exist for agents. Agents that usually do not change their location are called static agents, as opposed to mobile agents. The later have the capability to move from one machine to another. The case where they are sent to a machine where they perform their task and later are terminated or bring back a result, is called ’remote execution’. If they are allowed to decide where to go themselves, it is known as ‘migration’.
The characteristic ‘intelligent’ is getting more and more attention. An ‘intelligent agent’ is not just constructed to perform a single task, but has some (limited) knowledge and/or reasoning skills, which make it able to solve a whole spectrum of tasks or orders. As a further step, intelligent agents could be able to learn and adapt themselves to changes in their environment.

2.2 Agents within Distributed Management

With the networks getting larger and the methods to exchange information gaining more complexity, the human administrators are overtaxed. Some hope is put into intelligent agents used within distributed management.
With their help the automation of the elimination of errors could be raised to a higher lever. They are also expected to offer abstract views of problems, because they can do low level diagnosis themselves and prepare the information before they are carried on.

One way to integrate intelligent agents into the concept of distributed management is carried out in the earlier mentioned ‘management by delegation’.
A manager station is able to send functionality (including knowledge) as scripts, compiled programs in any form to other computers or network components, to solve problems, control an event or to observe certain conditions.
Hereby, the terminology is - again - not clearly defined. Sometimes the moved entity is called the ‘mobile agent’ that is executed on or by an ‘agent platform’ or an ‘abstract machine’. In other cases, the later is called the agent, which is said to be enhanced by the functionality it receives.
Whatever terminology is used, it is important to understand the difference to traditional OSI and internet management. It is no longer the data, that is moved ‘en mass’ to the manager; The management function, or at least a significant part of it, is brought to where the data emerges.
It is worth noting that a flexible form of this management by delegation exists where the managers and the agent platforms are put on equal levels and can work in both ways to achieve good co-operation.
More about intelligent agents in the context of network and systems management can be found in [MULL1] and [MAEC96].

A complex point of investigation, which is of need in any of the variants of management introduced (or similar ones), is by what means the functionality is exchanged and what level of abstraction or knowledge is expressed by them. This is the subject of the next chapter.

3. Exchange of Functionality

This section describes some spectra that exist for the exchange of functionality between entities of a distributed management system. It also goes into some aspects of what guidelines have to be followed, when developing a practical means of exchange.

3.1 Abstract View

Before looking at the means of exchange of functionality, it is important to have a look on the characteristics of what is exchanged.
It makes a great difference for the management system receiving the packet of functionality whether it only has to deal with independent units, which do their job more or less on their own, or also with packets that want to be closely integrated in sets of already received packets. This also makes a difference for the representation chosen for the functionality within the packet. While independent units may have an identical interface for their control, packets of the second variant may additionally have individual aspects that must be described separately.
Under the category 'what' is exchanged lays also the following: The functionalities may likewise contain knowledge that is needed to perform a special management task. Knowledge includes more complex descriptions of relations amongst (managed) objects and about the objects themselves, but also pure data, like single values and lists. The later are usually also implicitly included in program code, but the expected proportions influence the kind of representation that has to be chosen. It might be necessary to use a specialised language for more complex knowledge, which is capable of expressing relations and complicated descriptions.
Knowledge must not always just be an add-on to management functions. If the reasoning algorithms are fixed or known by all managers, they can - together with goals stating what wants to be achieved - be a full replacement for traditionally programmed functions. Such a pure exchange of knowledge also requires a mechanism to exchange goals that want to be achieved on the basis of the knowledge.

For the form of representation of functionality the possibilities can be chosen from a vast spectrum.
As already mentioned, an imperative language can be used to directly express functionality, while declarative or in some other way suitable specialised languages are preferably used to transfer knowledge.
For the former, script- or other interpreted languages, which are even readable to human users, could be used. Examples are Java-Script, SMSL (systems management scripting language) or 'SNMP script language'. Similarly readable representations of

knowledge exist in form of HP's 'Flipper Modelling Language' or, for example, parts of KQML. But, while these directly readable, interpreted languages are easier to be controlled by the management software, they might not be very efficient.
The other extreme would be to use (at least for the imperative languages) readily compiled machinecode. This is presumably the fastest, but brings disadvantages in security and makes the packets sent system-dependent.
In between lays the use of pre-processed, but still system independent languages. The Java bytecode is the best known example. The development of a bytecode for the declarative Flipper modelling language (FML) which carries knowledge in form of so called models is described later in this thesis.

3.2 Practical Issues

Besides the protocol used to co-ordinate the distributed management systems and to initiate and carry out the transfer of functionality, the means of representation of the later plays an important role. The protocol is considered out of scope of this thesis.
What means of representation must be chosen for a particular architecture or system is dependent on several conditions and cannot be decided in a general way. Of course the usual demands can be adopted: It must be practical, efficient and secure.

Practicality

Practicality in this context includes the possibility to express any required form of management functionality and to be able to bundle them in an appropriate way, as well as to divide large ones into smaller portions.
All this is generally met by common programming languages that are designed for multi purposes, especially because most of the newer languages use object oriented encapsulation mechanisms to provide clear interfaces to the code. This is valid for languages in their 'pure' textual form (like Java, C++, etc.) but also for their compiled object-code and bytecode, respectively.
On the other hand, if the chosen language is not really specialised for management purposes - not like the mentioned SMSL, for example - at least some libraries for basic management functions have to be provided. In addition functionalities expressed in these kinds of languages may not be automatically manipulated and otherwise processed by management-software as easy as this can be done with knowledge representation languages.

Efficiency

Efficiency plays an important role in two ways:
The carrying out of management with the help of the functionality packets must be efficient. This includes one-time tasks like decoding or even compiling when a packet is received or used for the first time and the use of computing time and other resources during each execution. Of course, interpreted languages always have a disadvantage in this regard, but this can again be minimised through specially pre-processed (bytecoded) forms.
The second efficiency consideration is about the typical size needed to represent functionality, which influences the network load through the traffic the transfer causes. The importance of the size is heavily varying with the expected rate of exchange and therefor with the whole management architecture.

Security

Security is an important issue in systems and network management, because a lack here can easily get a security problem in the whole managed system. Still, it is often silently ignored or wrongly assessed in existing architectures.
On the one hand, the transmission channel must in principal be secured by the protocol. On the other hand, there is no reason to trust a received packet of functionality, only because the transmission is considered to be secure. Digital signatures per packet, not per transmission, can help to decide whether packets from particular original creators are ignored, only partially trusted and, for example, only accepted after some additional tests, or are totally trusted, where the time-consuming tests may be omitted.

CHAPTER III - FLIPPER MANAGEMENT APPLICATION

1. Introduction into the Flipper Project

This thesis is concretely based on developments of the project at the Hewlett-Packard Laboratories in Bristol mentioned in the introduction in the first section. The main developments are the Flipper management application and the Flipper Modelling Language (FML) used to describe the environment's management by the former. All analysis within this thesis, even if they are of general nature, finally aim to the goal of the development of a bytecode for Flipper and the FML and its integration into the existing system.
Before the concrete analysis of bytecodes and the final development is described, this section introduces into the principles and purpose of the Flipper system and explains its position within the world of distributed management, as it was described in the previous section.
Readers interested in specific details of the implementation or architecture of Flipper or the FML will not find them in this section, but partly in appendix A (assuming they obey the restrictions of the HP confidentiality) and more comprehensively in literature available at the HP Laboratories Bristol, like [IZUR96], [FERR96], [NIEL96] and [COMB96]. For the understanding of the work in this thesis, it is not necessary to know these details.

1.1 Background

Already before the Flipper project was founded, the HP Laboratories worked on systems management. The Dolphin project, which was based on model based reasoning can be looked at as the predecessor of Flipper.
It was implemented in Smalltalk. Its goal was to investigate the principles of model based management solutions, which had hardly been analysed at that time. For this, Dolphin also received a kind of modelling language. Over the time, the prototype was more and more equipped with experimental features. Even possibilities of distributed management were already looked at. This is analysed in [MORE93].
While the researchers’ experience grew, the Dolphin prototype became a huge monolith, until it was time to collect the knowledge reached and to pack it into a smaller system.

This was the beginning of Flipper. It origins in the wish to realise systems management through model based reasoning with the help of a slim and powerful concept. The modelling language was redesigned, too and resulted in the Flipper Modelling Language.

1.2 Principles

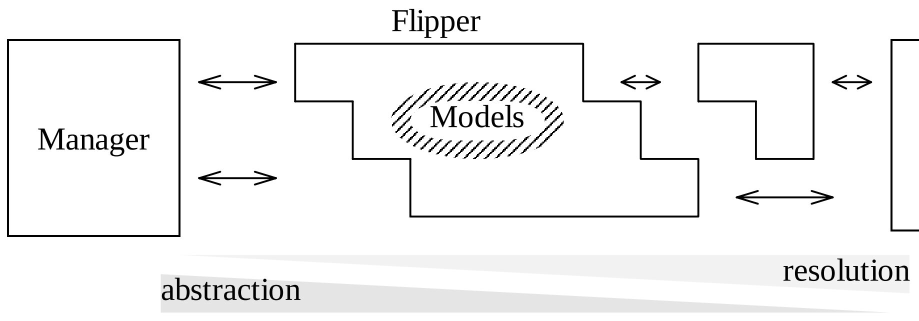
The goal of the Flipper project is to transfer the idea of model based reasoning to network and systems management and to realise this in an application.

Model based reasoning allows to capture properties of the managed world on various levels of abstraction and to use the resulting models to perform automated reasoning.
On a higher level one could model a computer network - as it is seen as a whole and describes what makes up its relationships to other things on the same level (other networks) and its dependencies to lower levels. The elements used to build this level would be modelled separately at a different place, and thus refine the description of the real world step by step.
Models may reach a very low level of abstraction. The lowest FML models finally base on modules programmed in different languages, which are closer to the system, like C. They take care of the accesses of the management system to the managed objects outside the system.
These transitions to the real world may be of various forms. According to the needs, they may be calls of informative functions of the underlying operating system or accesses of other management interfaces, like the one of SNMP agents.

From a certain point of view, two kinds of models may be distinguished. The first kind is restricted to the collection and evaluation of information. The corresponding underlying access modules also keep to this more passive role.
The second, active kind of modules partly uses and involves the first kind. Depending on the information gathered in the passive parts, active actions may be triggered. These are described in the active parts, where the underlying (now also active) access modules eventually perform or trigger the modifications on the managed objects.

The spectrum Flipper can be applied to is broad. This is of course reflected in the architecture (which is described in appendix A). On the one end of the spectrum stands the system administrator, which directly co-operates with the Flipper system.
On the other end there are single hardware components, which do not know anything about management comprehensions; They are controlled by Flipper through access modules. This end must of course not be fully exploited. Depending on the application it may be enough and make sense to build on existing levels of management, like stan-

dardised hardware drivers and routines of the operating system or more complex functionalities through already implemented management protocols or independent proprietary management applications.
Between the two ends lays the operation field of the FML models. They form abstractions of the low level objects, of their properties and their relationships. Step by step, higher levels of abstraction are reached, which finally are presented to the manager on the upper end.



Flipper within the spectrum of possible levels of abstraction

The level of abstraction the manager uses may vary. It mainly depends on the models in use.
The smaller element between Flipper and the managed objects suggests that in most cases it does not make sense to built the models directly onto the lowest objects, but to use - as mentioned above - already existing tools etc. as an intermediate layer. As a simplification, the access modules and other components of Flipper are not drawn into the diagram.

Although Flipper was originally designed for systems management, its application mainly depends on the models and of course the possibility to access the real world via access modules.
In principle Flipper could as well navigate an aircraft. A huge number of models could represent the various instruments (like for example the compass) and the machines (like the flaps or turbines). Other models would raise the level of abstraction to a degree, where the pilot can give simple commands like “Turn eastwards!” or “Hold an altitude of 1000ft!”. Flipper could - with the help of the knowledge contained in the models - find the right steps to perform the goal and trigger actions on the aeroplane.

The example already made use of the manager's possibility to give orders in the form of ‘immediate goals’ (“Turn!”) and of ‘persistent goals’ (“Hold altitude!”). In the later case Flipper additionally distinguishes ‘monitored goals’, like “Keep looking, whether the printer still works!”, which lead to notifications whenever the monitored state changes, and ‘resident goals’. These do not just monitor a special matter, but Flipper

tries to keep them satisfied, if necessary by triggering actions. The corresponding goal in the example of the printer would be “Keep the printer working!”.
The principle of separation used here matches the general one according to activity or passivity.
Additionally two kinds of the passive immediate goals are distinguished. Such, only interested whether a condition is true or false (or want to know at most one solution), and others that want to know all possible solutions. An example for the first kind is “Is there a working printer?”, whereas “Which printers work?” belongs to the second.

All models are programmed in the Flipper modelling language, while the programming languages C and C++ are currently used for the access modules.
Goals, given to the Flipper system by managers, are expressed in a subset of the modelling language.

2. Flipper as a Distributed Management System

2.1 Flipper in the World of Agents

Terminology Considerations

Flipper is considered to be an intelligent agent for systems management. The currently stable version is a stand-alone application, but some effort has been put into establishing the possibility for several Flippers to communicate and to finally build a distributed system in which Flippers can share knowledge and data to co-operate and function as a robust system.
The exchange of functionality is preferably based on models written in FML. This is analysed in the following section 2.2.
A different point of view on the terminology of agents is, as mentioned in section 2.2 of chapter II, not to call Flipper itself the ‘agent’, but to look at Flipper as the agent platform, where the exchanged models, goals or a combination of both are executed. These would be called the agents. As they are processed in a standardised way, similar to agents in script languages, for example, the idea of ‘remote execution’ fits to this terminology, too. As Flipper does not (yet) contain the runtime-management of its functionality-groups, except the possibility to add them, the term ‘agent-platform’ seems to be exaggerated. Therefore, the firstly mentioned terminology is preferred here.
Investigations of this subject have, for example, been carried out in [BDM96].

Facing the Problems of Distribution

When setting up a distributed system for general systems management, one has to face a heterogeneous environment. One solution, which can be found in other products, is to maintain independent and to some extent different applications for each system. These communicate via a common interface and therefore hide their true identity from the others.
From a business point of view, this is usually not bearable, so the Flipper team at the HP-Labs went a different way. The main Flipper code is written in C++, which is relatively portable, with the usual constraints of such a plan. But this is not enough to ensure that Flipper will run on most systems, because every system differs in the way resources are managed, network services can be used, etc. The access modules help to decouple the core functionality from the system dependent parts. (See section 1 of this chapter and the description of the architecture in appendix A.)

There is a second benefit of modular management applications. As not all systems can be expected to offer the same amount of resources an adaptable management system should be able to fulfil its (restricted) work on small machines with possibly only very little memory and computing capabilities. It should, on the other hand, be possible to use available resources at need. Modules to extend the management capabilities could help to adapt the application to the particular system it runs on.
The existing Flipper-prototype has grown beyond the expected limits and is too large to run on small computer systems or even at the side of devices like printers, etc. Considerations of making it more modular are being made, especially because some modularity that can be seen in the architecture is not completely realised in the application.
Belonging to the same category is the large compiler code built into Flipper. It is necessary, because the current Flipper deals directly with pure FML. In case this is going to be changed (also because of other, earlier mentioned reasons) to a bytecoded form, the compiler could be replaced with a comparably small (and faster) bytecode loader. Chapter V will use this fact as a motivation and explain the effects of the use of bytecode more in detail. From this perspective, the practical part of thesis can also be seen as a first step to reduce the size of Flipper's code.

2.2 FML as a Means of Exchange of Functionality

Models as Components of Functionality

Section 3.2 of chapter II explained the necessity of being able to group functionalities in a way that they may be exchanged in a useful way for management. It is planned for Flipper to use the FML models as a basis for this exchange.
These models can be put together in any way through FML objects and rules. This allows the model creator to find functionality groups that are small and functional enough to be suitable.
On the other hand, Flipper or the FML do not give guidelines how to achieve this, so model developers may look only at their specific problem to be modelled, but might not care, what this 'suitable' means for distributed management.
In fact, it is not easy to see how to group models and what belongs into one model. To put all objects belonging to one context together with the rules describing their relationships usually leads to very large models.
Another idea is to use one model per described object and pack the related rules with it. Nevertheless, it is not always clear where to put rules describing relationships between these objects, specially when these reside on the same level of abstraction.

An advice, which should be followed in any case is to distinguish between several levels of abstraction, what for example also means to keep auxiliary functions separate, and to have the scope of each model restricted to one of these levels.
A helpful extension to FML would be a kind of clustering suggestion. It would let the model developer list other models (besides their occurrence in the 'IMPORT'-section) that are used so frequently in the described model that it would be unwise to use it without having these other models on the local machine. This makes it possible, to keep models smaller than the functionalities usually sent and to regroup them as required.

Problems with Rules

Most of the dependencies between FML models in its current version originate from the use of rules that are defined in other models. According to the language definition, these models have to be listed in the IMPORT construct. The Flipper prototype does not check this yet. It only checks if used objects are defined in the imported models.
Flipper later does not even remember from which model a rule origins, but this could of course be fixed. The real problem related to this is of principal nature and not very easy to solve:
If the model uses a rule, which it does not define itself, it is still unknown in which of these models it is defined (assumed more than one is imported). This is clear for objects when the object type is referred to by its fully qualified name, but never for rules.
A similar reason does not exist for rules, for a very good reason. The use of an FML-rule does not uniquely call one particular rule, like a procedure call in other languages. It is wanted that a rule-use results in the application of several defined rules with the same rule signature (set of partial names) and matching types.
Note that because Flipper does not remember from which model a rule came from, it uses all matching rules. This is not even restricted to the imported models of the origin model of the rule-use.

In the end, this leads to the fact that a writer of a model cannot control or foresee, what is going to happen when his model is going to be used, because he can and should not know which extra models - besides his model and those listed in his import list - are loaded in a Flipper-system.
The following example helps to understand, how the evaluation of rules is affected. Language keywords are written in a non-proportional font. The syntax is 'shortcutted' where this is helpful.

```
MODEL A
      OBJECT AType
            … .         // object description
```

```
RULES
      [AType a] b IF
            … .         // rule body

MODEL NewModel
IMPORT A
RULES
      [AType v1] newrule […] IF
            [v1] b &
            … .         // rest of rule body

MODEL ExtraModel
IMPORT A
RULES
      [AType v2] b IF
            … .         // rule body
```

The writer of the model “NewModel” imports a model ‘A’. He knows this model and uses the rule “b” to evaluate one of his problems using the local variable v1. He also is sure about - and relays on - what the rule does. If his model is processed by a Flipper that owns this model and the imported model ‘A’ everything works just as the model writer expected.
What happens, if another Flipper loads the model “ExtraModel” in addition? When the rule “newrule” is executed Flipper searches for rules with the name “b” (with the matching parameter type) and uses all the results from all rules it finds after executing them. In our case, a correct rule is found in model “ExtraModel”, too. But the result of the rule-use from within model “NewModel” is now different from what the model writer expected.
Something similar happens, when the rule in the “ExtraModel” is a so called OVERRIDING rule, like in the following example, where additionally some types are changed, to make it work.

```
MODEL A
      OBJECT AType
      OBJECT ASubType ISA AType
RULES
      [AType a] b  IF
            … .         // rule body

MODEL NewModel
IMPORT A
RULES
      [ASubType v1] newrule […] IF
            [v1] b &
            … .         // rest of  rule body

MODEL ExtraModel
IMPORT A
```

```
RULES
      [ASubType v2] b IF
            … . OVERRIDING
```

The rule “`b`” in model “`ExtraModel`” overrides (substitutes) the rule in model “`A`” for parameters of the type “`ASubType`”. Now, the rule “`newrule`” does not get additional results, but possibly complete different ones.
The overriding rule is not only made available for other models, and the overridden rule is not simply used like types or other rules in the body of the rule, but the definition of an overriding rule changes the behaviour of another rule that is defined in a different model.
It is worth to note that this is not an analogy to the overriding mechanism provided by common object oriented programming languages. With these, there is a (sub)class that extends a superclass. If it overrides a method, for example ‘m1()’, then this does not mean the behaviour of the original method changes. It only means, that calling ‘m1()’ in the subclass invokes the new method (as defined in this subclass). The original method may still be reachable with exactly the same behaviour as before, using a special calling mechanism.
Unclarities or unforeseeable sideeffects should be avoided by the specification of the language. A possible solution is, to restrict the use of rules and of the overriding effect to the context of the model that performs the rule-use. In other words, to rules defined within itself and in the models it imports.

The problem is not severe and stays controllable at the current stage of Flipper’s development, where a Flipper runs as a single system and owns all the rules itself.
In a distributed environment, however, Flippers working together may be equipped with different sets of models. If one Flipper asks others for the execution of a rule, because it does not want to or cannot do this itself, the result may vary, depending on which Flipper finally carries out the work.

Need for Model Interfaces

A further problem is to some extent related to the question, where rules can finally be found.
Even the restriction suggested in the previous paragraphs by itself does not yet guarantee the correct behaviour in some cases. In cases where not all imported models are loaded into a Flipper, the use of rules may not lead to the execution of all possible rules. Those defined in the missing model would be left out, because a rule-use on its own does not say in which models the rule can (or could, depending on the parameter-types) be found.

The solution applied by the stand-alone Flipper is straight-forward: All imported models of all models must be loaded. This guarantees the presents of all possibly needed rules.
This is not practical for the distributed management environment, because the use of one single model would lead to the obligatory presents of many other models, down to the lowest level of abstraction. It is a clear obstacle for the unhindered distribution of models according to management reasons. Smaller machines could possibly even not cope with such a load of models.
If, on the other hand, a Flipper at least knows about the existence of the imported models and what rules and objects are defined by them, it is able to ask other Flippers for the execution of those rules it misses. But there is no need to hold the whole model locally.
Most programming languages provide exactly this feature through what is often called interfaces. They additionally help model developers to hide the implementation details from others, whereas the interfaces could be made public and enable the usage of the model from outside.
Such an interface could be introduced for the FML. The development of the bytecode in chapter V will show, that such an interface can also be generated automatically, so there is no immediate need to change the definition of the language.

2.3 Java-Flipper - Jiver

<u>Java in General</u>

One of the developments that attracted the attention of and brought changes to the whole world of the internet is Java, including the programming language, the bytecode and the virtual machine. It was originally developed by Sun for PDAs (Personal Digital Assistants, mobile phones and other ‘electronic helpers’. Later, it aimed for broader markets like interactive television.
Nowadays, Sun likes to look at Java as the programming language of the World Wide Web. This is mainly based on the fact that the two providers of web browsers with the biggest market share, Netscape and Microsoft, both very early equipped their product with a Java virtual machine. This is a platform for the execution of Java bytecode, to which Java programs are compiled.
The attention Java caused is not very much based on great new technological or theoretical novelties. It rather comes from the fact that there finally exists the possibility to write programs more portable than those written in one of the other widely used languages, and additionally to be able to integrate these programs easily into pages of the World Wide Web - which was booming at that time, too. All this was done in a way the biggest part of the internet only seemed to be waiting for.

Programming language related advantages of Java are that it is relatively easy to learn - what partly comes from its syntactical similarity to C++, combined with known, but in comparable languages rarely directly available features like garbage collection, multi threading and a well-designed error handling.
The portability is mainly based on the exactly defined virtual machine and the availability of standardised libraries for the access to the underlying system (like for example to files) and basic functions to build user interfaces.
All this results in a broad spectrum of usability of Java programs on many different systems, which is especially important within the existing heterogeneous computer networks of many companies and of course within the internet.

These advantages are exactly what the wish to use Java for management applications is based on. The management systems written in Java - or at least the front ends and general parts - must not be ported to and maintained on a number of different systems. Looking at the practical availability of virtual machines on all current main platforms one can see that the portability is not just theory. On the other hand, the practical live also brings some incompatibilities between Java or library versions, but to a much less extent than this is the case with for example C++.

Flipper in Java

Jiver is the attempt to make Java's advantages available for the Flipper technology, too. An additional point comes from the mechanisms for communication available as standardised Java classes, which Jiver may use for the interactions in the distributed case.
When porting Flipper to Java (where it is called Jiver) the team had the possibility to either use the opportunity for redesigning and reorganising the code or to do a simpler one to one porting. As they did not have the time and resources they went the 'cheaper' way, mainly to see whether the promised advantages can be met in practice.

The inference engine, eventually responsible for the reasoning on the models, and the internally used stores have been ported as they are. For the user interface, however, there was no direct way to map the different libraries of Microsoft C++ towards those of Java. A new simple interface was developed instead.
Still missing in Jiver is a FML compiler. Modifications on the compiler of Flipper, which generate a fixed set of Java classes from FML models, are used as a work around for test purposes. Except basic test modules, no access modules have been ported. Therefore, Jiver stays a pure research tool for the moment.

On the other hand Jiver also contains some extensions in comparison to Flipper. It has already been used to test some aspects of distributed management. For example, it has a module called Jins (Jiver Name Server) used to control the communication among Jivers and with the machine trader.
The machine trader administers several Jivers running on the same machine as itself. It provides a quasi-transparent access to other Jivers and hides whether the communication is performed locally or to a remote computer.
The first machine trader in a network (the one that cannot find another one that is already running) gets a special duty. It is the top of the tree-like organisation of machine traders that is installed when more of them are started in a network. The resulting static tree is of course not really suitable for real world applications, because it is too vulnerable and therefore insecure, but for now it serves as the basis for tests.
Additionally, the distribution is currently restricted to the delegation of rule evaluation. All Jivers must posses exactly the same models to avoid unforeseeable effects. It is not possible to distribute knowledge.

All in all, Jiver can be seen as a first attempt to combine the advantages of Java with Flipper’s model based reasoning qualities for distributed management. It is not completed and serves as a platform for research. More about Jiver is documented in [CERF96].

3. Additional Security Aspects

The third part of section 3.2 of the previous chapter already informed about some security aspects concerning the exchange of functionality between management stations and stressed their importance. All this is absolutely valid in the context of Flipper, too.
An additional aspect origins in a very different point of view to management. Companies want to secure the knowledge they put into management systems from others. The Flipper-system offers - in form of the Flipper Modelling Language - a means to express knowledge needed for management. In a possible business scenario, other companies may produce their own models, which they may make public to enable Flipper to manage their products. But these companies still do not want to make these models easily understandable for anybody else.
A scandal with Microsoft's Visual Basic showed how concerned companies react. Somebody found out that the program file generated by (the meanwhile outdated version of) Visual Basic keeps every detail of the readable textual program code, like variable or procedure names and even comments. Many companies developing applications with this Microsoft tool were upset and felt that their innovations were in danger, because the generated program files were too easy to analyse and understand by other parties.
For the bytecode for the Flipper Modelling Language it makes at least sense to leave out the comments and local variables. The names of objects and rules must be reachable from outside, so it makes no use to change them. An exception would be a model that is used from within one company's models only and where its contents must not be made public. An additional exception would be non-public objects and rules, but these concepts do not yet exist in FML.

CHAPTER IV - BYTECODES

The previous chapter III described various problems originating from the context of Flipper in the world of distributed management and from having a look on other related subjects. Different points of these investigations showed that the introduction of a bytecoded form of the Flipper Modelling Language can help to solve or at least reduce the effects of certain difficulties.
The development of such a bytecode is one of the main goals of this thesis. The first section of this chapter first of all clarifies, what is meant by bytecode and which different kinds of bytecodes exist. It also shows that they have their own justifications in different environments.
The second section introduces two existing bytecodes. It investigates what characteristics may be helpful for the bytecode for the FML, too.

1. Kinds of Bytecodes

The Term ‘Bytecode’

Originally ‘bytecode’ only means that ‘*something*’ is coded on the level of bytes. A ‘byte’ in computer science usually stands for a unit consisting of eight “bits”. Bits may assume one of two possible states, often represented by ‘1’ and ‘0’. The term bytecodes is often associated with a form of representation that is hardly readable or understandable for humans or only by the help of special tools.
When technical literature speaks of bytecodes, this ‘something’ commonly refers to some kind of programming language. Apart from that, ‘bytecode’ can also be found in the context of the description of file formats.
The term ‘coded’ in the upper paragraph is used in its usual meaning. Without giving an exact definition, it at least means that there exists an understandable or verifiable transformation between the original form (for example the programming language) and the resulting code. The transformation must not be injective, so information may be lost during the process of coding. It also does not have to be completely deterministic. However, generally a bytecode is expected to represent the original (programming language like) form in a way that makes it useful for the purpose the original form was intended for, too.
This by itself would not justify the introduction of bytecodes. These are furthermore used to achieve additional goals and to accomplish the original ones more efficiently. Examples for this are a reduced consume of memory (at runtime and when storing),

direct usability (to save pre-processing steps), improved security and speed at execution time and so on.
One has to mention that the term ‘bytecode’ - like ‘code’ in general - is used in two different variants. On the one hand it labels the format and rules how the coded form is built and what it has to look like. On the other hand, the result of a particular process of coding is also called a ‘bytecode’.

Structures and Forms

The following passage discusses the structures and forms of bytecodes.
A bytecode as a whole may be looked at as one large data structure, which is kept in files or simply in the memory of a computer. Embedded in this large structure are smaller units of some form that give information about the whole bytecode, about smaller fields within the structure or they simply contain data.
One special way of organisation and meaning of these fields is so commonly used that it is very often equated with bytecodes. It is very similar to a machine code, where each field (usually only few bytes in size) represents an instruction that is understood and may be executed by an interpreter. The later may be realised in hard- as well as in software.
An example for hardware interpreters are microprocessors. Their machinecodes would here be called bytecodes. A simple interpreter of the second kind is the MI-interpreter commonly used at universities. It eventually emulates - with some restrictions - the processor and memory of a VAX computer.
More complex software interpreters are the various kinds of BASIC interpreters, which are still widely used. Most of them do not work on the BASIC text, but on a to some extent pre-processed bytecode. The structures of this code are much more complex than those of the previously mentioned machinecode. They contain more fields with extra information describing bytecode internal arrangements and a recursive relationship of containment among fields is possible, too.

Having a closer look at the examples of BASIC and the machinecodes, one recognises different characteristics concerning the level of abstraction. The instructions of a machinecode are more or less elementary and therefore comparably easy to execute for processors. It is not necessary to analyse the instructions themselves in a complex way, because their composition is not variable or only in a very restricted way.
In opposition to this, many BASIC instructions are coded on a higher level. The transformation of a simple ‘PRINT’, which prints text or the contents of program variables to the screen or a printer, into a machinecode would lead to a number of different instructions, which may even vary according to the special form and composition of the ‘PRINT’.

The higher level also comes from the fact that fields of a BASIC code may be interlocked just like the original BASIC language constructs. These complex relationships are not simplified for the bytecodes of BASIC, but represented as they are.

In some cases it may even make sense to go away a further step from the exclusive representation of elementary instructions within the bytecode as this is done in the BASIC bytecodes. This may eventually lead to a complex, recursive and intertwined data structure.
Such a kind of bytecode may be especially suitable for the representation of declarative languages. These languages purposely avoid to describe how something has to be executed. Besides the resulting advantages for the process of programming, they leave the choice of the algorithm (the ‘how’) at the interpreter. It sometimes does not choose the algorithm until it is really needed at runtime. The choice then may depend on other circumstances and the program data.
Methods exist, to translate declarative languages into an imperative form. These could also be used to develop bytecodes for those languages that are composed of primitive elementary instructions. However, then one has to recognise the fact that the information of the ‘how’ has to be included. This might be useful in some cases, for example within interpreters, which - for efficiency reasons - internally generate a bytecode similar to machinecode and work with it at runtime. The choice of the exact algorithm would have to be made anyway - at least while interpreting - so there is only little loss of flexibility in this case. An example of such an internal bytecode is presented in section 2.1.
In other cases, where aspects of data exchange, size and so on are more important, it might be of advantage to keep the bytecode in a somehow declarative form, too, and to orient its format on more complex data structures. Depending on the complexity of the language, this may lead to a more complicated interpreter, because it has to analyse more and to decode more complex fields as compared to the mainly sequentially ordered instructions of a machinecode. This is opposed by the advantages of leaving the choice of the algorithm outside the bytecode, to be able to use it in different ways.

Both possible kinds of bytecodes, as explained above, have different main focuses and aim towards different goals. As usual the specific advantages and disadvantages have to be investigated individually for every case. Section 3 will also go into this, for the special case of the bytecode for Flipper and its FML.

2. Investigation of Existing Bytecodes

This section investigates two existing bytecodes, to reuse parts of the experiences that have been made with their development for this thesis and to get an overview of the complexity of parts of the problem space.
The following two bytecodes differ in their origin, purpose and usage. Various parts of these bytecodes influence the development of different parts of the Flipper bytecode. Other parts are not relevant, due to the different circumstances. These are also described to a less extent.

2.1 Internal Bytecode of the Interpreter of 'Gofer'

Gofer is a programming language, which is used at various universities, mainly to introduce students in basic programming techniques without the need to teach the understanding of one of the imperative languages. Additionally it is used within a long list of researches for theories about programming languages, (automated) prove theory and a lot more.

Introduction into Gofer

Because Gofer is not so widely known as the later described Java, this section gives an overview of Gofer without going into the syntax or exact details of all realised concepts.
It is not necessary for the reader to be familiar with all peculiarities listed. A basic knowledge is enough to follow the main goal, the investigation of the transformation of the declarative (functional) code into an imperative bytecode. A comprehensive description of Gofer for interested readers is [JONE95].

Gofer's main qualities are:

- purely functional language,
- non-strict evaluation,
- higher-order functions,
- extended polymorphic type system,
- user definable type-overloading,
- type classes with multiple parameters,
- user definable algebraic types and type synonyms,
- pattern matching,
- list comprehensions,
- contexts within the definitions of types,

- basic input/output facilities and
- strongly syntactically and semantically similar to the widely used language ‘Haskell’.

Gofer is purely declarative (functional) language without added object orientation or imperative aspects.
Its special strength is an outstanding, complex - but not complicated - type system, which goes beyond usual possibilities. This is partly related with the non-strict lazy evaluation, where expressions are not evaluated, until its explicit result is really needed. This makes the existence of constructions possible that are not explicitly known in other languages, like infinite lists, undefined states and many things more.

The Gofer System

Gofer is not only the name of the programming language, but of the whole interpreter and environment for the interaction with the user is called like this. Language and system have been developed in the same go by Mark P. Jones.

The most important aspects that have been in mind during the development are the following:
The Gofer system was, from the very beginning, designed for extendibility, because the final goal was not really clear yet and the Gofer language should - to some extent - stay compatible to Haskell, even if the Haskell standard changes.
As it was primarily intended to use Gofer at universities and at student’s computers it has to be portable, simple to use and small enough to run on small computer systems. Also, enough performance is needed to be able to cope with the complex type system and the resolving of expressions during the interpretation, so that Gofer can still be used as an interactive system.

A very important and theoretically interesting part of Gofer is its type system. However, a comparable type system does not exist in the Flipper Modelling Language, so it is of no further interest within this thesis.

The Gofer Evaluator

Of more interest is the internal bytecode of Gofer, which is created from an input in textual Gofer right before the interpretation. Most of the facts about the bytecode and the virtual machine are described more in detail in [JONE94].
A speciality hereby is the process of transformation, which may be used in Flipper in a similar way in the future, when possibly a direct interpreter will be designed for the FML. The original design decision of Flipper to run on small systems underlines these similarities.

To make the purpose of certain actions of the transformation steps understandable, the final instance interpreting the bytecode is roughly described beforehand.
This instance working on the generated bytecode is an abstract machine with a virtual memory area containing the bytecode and a stack containing the expressions to be evaluated.
All it does is to take the expression on top of the stack and to evaluate it. If the expression is a function, then more elements directly below the top of the stack may be involved as parameters to the function or as local variables. The function - with the exception of built-in C functions usually expressed in bytecode - is called.
To work with values of parameters the abstract machine has one single register. This can be interpreted as any of the Gofer internal types, like integer or float. Before a called function starts with its own calculations it usually uses the ‘EVAL’ instruction of the abstract machine to load the register with the value of the first parameter, which is represented by the expression on top of the stack right after the function call.
In the other case it is a simple value, which is loaded into the register. But again, the expression may not be readily evaluated, in other words it again is a function so it has to be called before the rest of the first one is processed.
In the end, a single expression that must be a value remains on the stack. It is the result of the evaluation.

For this method to work it is required that the complex expression represented by the stack at the beginning of the evaluation is in the right form. This form is achieved by the same transformation of Gofer textual expressions that is used for the generation of the bytecode for Gofer functions. Only an additional step is required for the later, which translates the expressions into instructions of the abstract machine.

To be able to operate on only one type on the stack, the internal type ‘Cell’ is used. A ‘Cell’ can be popped, pushed, deleted or exchanged with another ‘Cell’ and so on. The value of a ‘Cell’ can be interpreted at need as anything used within Gofer, like a constant value, a Gofer pair, a variable name or a type class.
The correct type of a ‘Cell’ at runtime is guaranteed by the type checks performed when the bytecode is generated .

Generating the Gofer Bytecode

The Gofer bytecode is generated in four steps called:

- lexical analysis,
- static analysis
- type checking and
- compilation.

All steps are kept in their own code modules. Therefore, the generation of the bytecode stays modular and is open for future changes.
The following explanations of the steps are simplified on a general description and on those parts also interesting for considerations of future changes for Flipper.

The first step is performed by the parser. It finds syntactical errors as usual and eliminates some language constructions that have an identical meaning as another one with a different syntax. The input is eventually transformed into an abstract syntax, which is still very similar to the original language, but without 'syntactical sugar'. Additionally the parser generates lists of expressions found on the top-level, like for example user defined types and type synonyms. It also sorts the newly defined operators by precedence.

This is followed by the static analysis. It performs simple semantic tests, like checks for overlapping areas of validity of variables or types with the same name, which occur after re-definitions of variables, and tests for the correct format of expressions.
Some lists, partly already generated during parsing, are completed. Constructors of types, for example, which are only implicitly expressed through type definitions, are added.
To prevent the maintenance of lists of constants and the resulting extra time of dereferencing symbols for them at runtime they are replaced by their values. Further, all definitions are ordered according to their dependencies.

The most complex step is the type checking. As Gofer's type system is extraordinarily powerful, the type checking, or even the attempt to give a comprehensive overview is by far too complex compared to its use within this thesis.
It is important to know that the type checking guarantees that all types of expressions used at runtime are correct under any circumstances. This gives a better performance for the abstract machine, which does not need to perform any more tests, whether a value or expression used from the stack - like the parameters - or the content of the register is of the correct type. It also saves space, because the abstract internal representation of the Gofer program may now be transformed into a simpler form, which does not contain the superfluous type information.
Even if the type system of the FML is comparably simple extracted type tests are able to help Flipper in just the same way.

The last step for the generation of the bytecode is the compilation of the now simplified (tree-like) program representation into instructions of the abstract machine.
This is achieved through the transformation into supercombinators. A supercombinator is a closed global function, or in other words a function that has no free variables and is defined on the highest level of a program.

The first phase of this transformation simplifies the remaining complex constructions, like case and lambda expressions or monads, by creating a number of local functions and adding local variables and calls to built-in primitive functions. The later helps to solve expressions using Gofer's pattern matching.
With this phase all free variables are automatically eliminated. In the next, the so called lambda lifting raises the original and newly generated local functions to global top-level functions.
Resulting the steps so far is a list of functions only connected through calls. They are independent in any other regard and only consist of constant expressions, the stated calls and of the use of primitive built-in functions.
Due to this independence of the individual functions and the simplicity of the parts they are composed of, all this may now be translated into the instructions of the abstract machine. These are on a comparably low level and mainly serve the manipulation of the stack or regulate flow control.

The Gofer bytecode is only defined to the described level of machine instructions. The bytecode is only used internally and not stored or exchanged. Therefore, there is no need for an exact definition on the level of bits or bytes. Different compilations of Gofer systems on different machines are allowed to use their own orders of bits within a byte, for example.
Note, that this is not what a distributed Flipper system using exchangeable bytecodes will be like. There, the definition of the bytecode must go into the very detail of the low level representation.

2.2 Java Bytecode and Java Virtual Machine

Section 2.3 of the previous chapter already introduced the attention Java attracted in the world of the internet.
As mentioned, a great part of its success comes from the precisely defined virtual machine, which to a great extent guarantees equal behaviour at runtime of the programs in Java bytecode on any system it is implemented on.
Several advantages are expected of the introduction of the bytecode, which can be looked at as an intermediate layer separating the Java language from the Java virtual machine. These are described in this section, as well as the problems and challenges involved by its development.
Again, a comprehensive analysis is too complex - measured by the purpose of this section. So, the focus is set on general aspects and on those parts relevant for the development of the Flipper bytecode.

The Java programming language is already well known and good introductory literature, like for example [LEPE96], is available. Additionally, it integrates techniques like object orientation, controlled multi-threading and so on, but does not offer any completely new concepts. Therefore descriptions of language details are omitted.

A Java virtual machine deals with Java bytecode, so nearly no transforming steps are necessary. They are very much related and as a result not described totally separately. The bytecode loader offers some interesting features, which are explained with the idea of a bytecode loader for Flipper in mind. More details about the Java virtual machine are described on [LIYE96].
On the other 'side' of the bytecode the compilation of textual Java code into this machinecode-like form follows the known principles of translating object oriented imperative languages and brings no new ideas in the context of this thesis. See [HEFF95], for example, for the compilation of imperative languages.

Data Types

A different approach for the representation of the native Java types than Gofer's multipurpose base type 'Cell' has been chosen within the Java bytecode.
Just like for hardware architectures, a basic unit called 'word', which is a group of a constant number of bits, is used as the basis for all (independent) types within the data areas of the virtual machine. The standard does not prescribe the size of a word in bits, but demands that all types except 'long' and 'double' have to fit into one word, the former into two. Nevertheless, a 'word' is not a basic type.
Like 'long' and 'double' the other types 'byte', 'short', 'int', 'char' and 'float' directly represent their equivalents of the Java language. The type for characters, for example, must be able to represent the whole Unicode and therefore be at least 16 bit in size. The range for the values of the other types is defined precisely, too. Java's boolean type does not exist within the bytecode. Except in arrays of bool, which are treated separately, the compiler substitutes it by an integer.
Three additional kinds of reference types help to administer the more complex types of the Java language: 'class type', 'interface type' and 'array type'. Each of them may take the value of the 'null type', representing the Java 'null'.
An extra type is added to the types of the Java language: 'returnAddress' is used to store return addresses.
Note that the unit 'word' is not applied on areas where the program code of the bytecode is stored.

Similar to Gofer's bytecode, values do not contain any extra type information. Java does not allow to reinterpret values in memory as different types, like for example C allows to use an integer as a pointer, but the type of a value is implicitly contained in the instruction operating on it. So, a Java-Compiler has to do type checking, to guarantee the correct behaviour.

Data Areas of the Virtual Machine

Like conventional machines, the Java virtual machine maintains different data areas for different purposes. All of them are taken from the large basic area, which is as usually called the heap. It is used for arrays and the data of class instances.
Another big part of the heap is used for the ‘method area’. It contains class related information like class data (constants and variables) and also the code of the classes’ methods. Constant values taken from constants of the original Java code or internally used ones, like the symbol table, are stored at a common place within the method area, in the so called ‘constant pool’.
Other data areas must not only exist once, but per thread. These are the program counters, stored in the virtual register ‘pc-register’, and the thread stacks. An additional stack may be created at runtime for native methods, which are implementations of methods written in a different language than Java (most commonly C or C++) and present as native machine code.

Virtual Machine Instructions

For efficiency reasons, instructions of the Java virtual machine are only one byte (eight bits) wide. This drastically reduces the number of possible instructions and remembers of RISC-architectures, whereas the overall concepts of the virtual machine (it has, for example, no registers to store temporal values) are more like from a CISC-architecture.
The instructions as well as their operands are not adjusted to special addresses, like the two or four byte borders typically used on 16- and 32-bit machines. Eventually the code of a Java method forms a compact one byte stream, which is defined independently from particular computer architectures and therefore offers a certain independence of the underlying system.
The set of instructions is not designed in an orthogonal way. For example, not all arithmetical (and logical) operations exist for all arithmetic types. They have to be replaced by combinations of conversion instructions with existing arithmetical (or logical) operations. To reduce the number of instructions and the complexity of the virtual machine even further the smaller types ‘byte’, ‘short’ and ‘char’ are converted into an ‘int’ when stored on the stack, which is also the storage place for all local variables.
Generally, no operations are defined for them, except for explicit conversions and for arrays with their type for the elements. The use of integer as a substitution would here lead to too many disadvantages, mainly concerning space. Boolean arrays in Java are replaced by arrays of byte.

All in all one can see that the one-byte instructions are a good compromise between compactness of the code and the speed of the process of interpretation.

Besides the stated arithmetical (logical) operations and conversions, the following large groups of instructions exist:
The load/store operations include those for the manipulation of the local stack. Their number is reduced by the fact that no temporary registers exist, which otherwise lead to a number of logically identical instructions working on different registers.
Flow control, including the call of methods, is solved by the usual instructions, like 'JUMP' etc.
Not found in the instruction set of most microprocessors are possibilities for object manipulation and exception handling. Instructions for the synchronisation of threads also belong to this especial group.
An interesting possibility for optimisations at runtime is the special '_quick' instructions. They must not be included in a bytecode but are inserted by the virtual machine itself. This is often used for tasks that have to be performed only once per method, for example. To achieve this, a '_quick' instruction is added at the top of the method, followed by the special task. When it is executed for the first time the virtual machine effectively replaces it by a jump to the address right after that block. The task is no longer performed at following calls of the method, and no extra flag with a conditional jump is needed.
An exception for the one-byte length restriction is the 'wide' instructions. These are variants of other normal ones, which are preceded by the opcode for 'wide' and therefore eventually more than one byte long.

File Format

The definition of the Java bytecode does not only include the form of representation of the instructions described above, but exactly prescribes the format of the file keeping the bytecode for storing or transfers.
One Java bytecode file contains exactly one Java class, though the files are also called class files.
It is devided up into several sections occurring in the same order as listed below:

- file header,
- constant pool,
- access indicator,
- entry for the name of the class and superclass,
- interfaces,
- fields with attributes,
- methods with attributes and
- class attributes.

The header, the access indicators as well as the field for the names have a constant size. The others are preceded by a length field.
The *file header* contains the usual information. A 16-bit magic number (with the hexadecimal value 0xCAFEBABE) identifies the file as Java bytecode. The version number of the Java compiler follows. It is more related to the version of the bytecode definition the compiler is able to built than on the product version of the compiler. Therefore it makes sense to look at it as a version number of the bytecode standard. The number is devided into a major and a minor number, where changes in the former reflect profound changes bringing along incompatibilities, while new minor numbers are used for little bugfixes and add-ons. A virtual machine designed for a specific version number usually is able to cope with all bytecodes carrying the same major and the same or a lower minor number. This opens the bytecode for future changes, and makes these changes easily detectable for older Java virtual machines.

The *constant pool* of the bytecode corresponds to the one of the virtual machine. It is the result of a very basic design decision, to collect all constant expressions, like names, number and indices at one place. They are referred to by a kind of bytecode-internal pointers, which are two bytes in size.
All entries in the pool have a one-byte identifier for the entry type at the first position. The rest of the entry is of variable length. For some, the length is implicitly contained in its type but for most an extra field for the length exists. Entry types are defined for constants of the numeric and character types of the bytecode, for so called method references (used in Java interfaces) and for class names, fields and methods.
The three entries described at last do not contain constant names themselves. Like string constant entries they refer to another entry containing a string coded in the UTF-8 standard.
This standard allows to use characters with a nearly unlimited size. The Java bytecode however uses - like the Java language - only 16-bit wide Unicode characters, which need at most three bytes in their UTF-8 representation. This price is easily paid, knowing that the most commonly used range of characters (the ASCII-characters) - which are identical to the values zero to 127 of the Unicode characters - take only one byte. These string constants may be referred to from various parts and may therefore help to save some space.
A further design issue - which similarly has to be taken for the rules in the FML bytecode - is the representation of methods and their parameter or return types. This is solved through strings combining the name of the method with a very short form of the types. Intuitive letters like ‘i’ or ‘f’ for ‘integer’ and ‘float’ or a bracket ‘[‘ for arrays are used.
All in all the constant pool does not seem to be designed for space efficiency. Every small pool entry contains an identifier. Several entries, like the one for Java strings, only contain this identifier and a reference to another (for example UTF-8) entry. For

strings this leads to four extra bytes - not counting the necessary value of the string length itself.
An order of the fields, sorted by their type, could help to reduce this overhead. This would, for example, lead to a kind of header saying that twenty entries of type integer follow instead of storing the identifier ‘integer’ twenty times.

Stored in a bit field, two bytes in size, the *access indicators* inform about accessibility (public, private and so on) and other attributes of the class. An important attribute, for example, says whether the bytecode describes a full class or actually just a Java interface.
The *name of the class* and of the direct *superclass* - remember that Java allows only single inheritance at this point - are indices into the constant pool, where the real names can be found.
The following area lists the direct *super interfaces*. Together with the three preceding small fields, it exactly sorts the described Java class into the class and interface hierarchy. This again shows, that the bytecode is independent of the Java language, but nevertheless strongly influenced by the Java language definition. Other languages could of course be translated into Java bytecode, but the effort will be higher in most cases.

The following areas for *fields* (which is the Java term for class and object variables) and *methods* also have access indicators similar to the one described above. They additionally consist of indices to their names and types within the constant pool.
Both areas can and sometimes must be extended by attributes. These may be defined by vendors, to provide special debug information, which is, for example, only understood by a the virtual machine of the Netscape Browser. Other virtual machines that do not understand the attributes must ignore them. This does not lead to any further problems because unofficially added attributes must only contain additional but never semantic information.
Examples of officially defined attributes are: Constant and initial values for fields (again realised as indices into the constant pool), a list of possible Java exceptions that a method may throw and even the program code belonging to methods. The later two are prescribed to be present exactly once per method.
The code attribute is the most complex one. Along with the code - a sequence of instructions of the virtual machine - it contains the maximum number of local variables and stack space plus a table of the code areas secured by exception handlers associated with the specific type of the exception. It may contain attributes itself, for example, for the names of local variables and line numbers.

The last area contains the *class attributes*, which are definable by vendors just like the attributes above. Officially defined attributes contain debug information, like the filename of the source file.

Verification of the Bytecode

Whenever a Java virtual machine receives a new bytecode it is not simply accepted. Complex verifications are performed, before it is used. Although the Java compilers perform checks on the Java program, like the type checking mentioned above, and usually compile it into a correct bytecode the tests in the virtual machine are carried out for good reasons.

The developers of Java want it to become widely used in the World Wide Web. However, in case it would be possible for Java applets to uncontrollably access the machine they are executed on, nobody would allow them to run on the local machine - assumed they did not write them themselves. The web browser - or more exactly its virtual machine - must restrict the possibilities of the Java applets at best in a user configurable way.

Generally it is not too complicated to control correct programs. As the virtual machine provides an exact interface to the host computer, a selective activation of allowed parts of that interface leads to a certain control over the Java programs. Other Java bytecodes on the other hand that - possibly on purpose - do not reflect a correct Java program or have malformed entries may cause unforeseeable effects in the virtual machine and finally on the computer system.

In praxis, it is hard to program a virtual machine that is immune against such bytecodes and even harder to prove that it is. It is easier to create a correct virtual machine with the assumption it only has to work with correct bytecodes and as a preceding step to check the received bytecode for correctness. This approach usually brings a better performance concerning the speed at runtime, too.

A second aspect that requires the tests is that incompatibilities between different Java classes or packages must be detected. These usually come from differences between the version expected and the one currently loaded by the virtual machine where for example parameter types, the type of a return value or the accessibility of methods have changed.

Additionally a bytecode may be corrupted through errors in the filesystem or during its transportation. Many simple falsifications can already be detected at the first step of the loading, like for example a wrong entry for the length of a following area. Others, on the other hand, are very hard to detect, especially when they are done on purpose and are no violations against basic rules of the structure of a bytecode.

To provide a secure virtual machine received bytecodes are tested in various stages. For these, the fundamental design decision for Sun’s implementation of the Java virtual machine was, to move as many tests as possible from the runtime towards the binding of the class of bytecode into the runtime system. This is not a must for virtual machine implementations but is followed by most other vendors as well.

As the following short explanation of the four test phases shows, some of the tests are very complex and could, like for example the maximum stack size, be observed more easily at runtime. Still, the higher performance when executing the program together with the fact that this also replaces some runtime errors by easier to handle load-time errors is assessed as more important.

The first phase already takes place at loading time. The magic number, the right file length and matching length of the attribute fields are checked. Also the constant pool is superficially checked for not allowed entries.

The next two phases are performed at binding time of the bytecoded class.
Firstly all tests possible without looking at the program code in the code attributes are performed. Examples are tests for correct names and type identifiers or that the superclass is not a 'final' class - meaning that it must not be subclassed.

The most complex tests take place in the third phase. Flow analysis is used to prove that whenever an instruction is executed - no matter how it was reached - the following conditions are met:

- The parameters are of the correct types,
- the stack is in a correct state, and
- no local variables are used that have not yet been assigned a value.

Furthermore, all instructions must be complete (for example, no parameters are truncated by the end of the code attribute) and the destination of jumps valid. Also, the execution of instructions must not be able to "fall off" the code attribute and method calls must be correct as well as all indices to local variables.
When performing the tests it is avoided when possible to load other classes that have not been loaded yet. The assignment of an expression of type 'class A' to a variable of exactly the same type is correct in any case. The class 'A' must not be loaded to check this. If, however, the variable is of type 'class B', then both classes must be loaded to be able to check that 'B' is a superclass of 'A'.

The last phase is only implicitly present within the execution of the instructions of the virtual machine. It includes the test of matching access rights and triggers the loading and initialising of other classes. This is one of the possible uses of the '_quick' instructions described further up. To optimise the runtime behaviour certain tests that must only be executed once are partly implemented by these '_quick' instructions.

It is important to see that the tests performed by the virtual machine are immense. The complexity of the loader is comparable to or even exceeds that of a Java compiler.
For the development of the bytecode with loader and generator for Flipper this warns to keep an eye on the complexity of the loader resulting from design decisions for the bytecode. As Flipper wants to have a small loader replacing the larger FML compiler, the "weight" must be moved from the loader to the generator, without loosing the advantages of the tests at load-time.
This is of course discussed more in detail in the next section.

CHAPTER V - DEVELOPMENT OF A BYTECODE FOR FLIPPER

The main task of this chapter is to develop a bytecode for Flipper and its modelling language. The bytecode should fulfil the wishes derived from Flipper's move towards distributed management and all the others mentioned further up in this thesis.
The important point is to integrate the new ideas into the existing project - with its architecture, language definition and also the implementation.
With some objectives the thesis goes even further. It tries to foresee future changes and makes certain choices in way that the added features will not block them. Additionally, new features are added to Flipper at some points where it seems appropriate and achievable in the context of this development and integration of the bytecode.

1. Requirements on the Bytecode

The two main goals of the development of a bytecode for the Flipper Modelling Language are:

- The bytecode may serve as a kind of link between a model compiler and a Flipper. When this is achieved, the first step towards a separated compiler - and therefore a smaller Flipper application - is done.
- It may also be part of the links among several Flippers during management, where it serves as an efficient basis for the exchange of functionality.

The goal to completely separate the compiler cannot be achieved yet, because the compiler is still used to support too many different purposes within Flipper. The monolithic architecture of the compiler does not easily allow to split these purposes from the full FML-compiler, but this is not investigated explicitly in this paper. The implementation in chapter VI is partly influenced by this circumstance as well.
For now a precondition to be able to separate it later is worked on: A well-defined means of representation for the pre-processed form of the models, which can be generated by the compiler extended with a bytecode generator, and later be loaded into Flipper with a comparably simple loader.
The second point must not be confused with the work on an inter-agent protocol. The bytecode is the basis of model representation for the exchange, but it does not prescribe anything for the exchange itself.
A third but for the moment less important goal is to shield the Flipper application from certain changes in the FML. Profound alterations of course, like the planed meta level,

must be reflected in the bytecode as well; Still, the bytecode functions as a means to decouple the FML from Flipper.

While achieving these main issues the development of the bytecode is not only about finding *some* representation of models; a well-formed bytecode is able to offer extra advantages that are hard to gain by the purely text based FML.
This section explains what the requirements on the bytecode are to be able to say it is well formed in the sense of fulfilling the former stated goals while adding other advantages. These requirements are mainly derived from the investigations in the previous chapters and only described briefly here to have them in a central place, to be able to refer to them and to make sure the reader understands them in the same way they are understood in this document. This section does not yet explain how they are met.

Independence from the Underlying System

The first requirement is "*independence from the underlying system*". It is derived from the wish to use distributed Flippers in a heterogeneous environment in the future. At the moment only the version for Windows NT is in use, so this is not yet a problem. In these heterogeneous environments however, you have to face different methods to code bytes or characters, and worse: low-level models and underlying access modules may be useless or have to be available in different versions.

Enable Development of an Efficient Loader

The second requirement is, to design the bytecode in a way, which later makes it possible to design an *efficient loader* for Flipper. Efficiency here is meant in two ways: *regarding size and speed*. As models usually are loaded by far less often than processed, the importance of the second meaning highly depends on the frequency of loading (and in future versions possibly also unloading) of models. This would be closely linked to the form of distribution and the kind of interaction between Flippers. The size of the loader is a more constant factor, but nevertheless very important, because the presence of a loader (however simple it may be) is hard to avoid. It is known that some work (e.g. certain checks) could be taken over by the inference engine at runtime. This may help to reduce size and improve speed of the loader, but as it is obviously not a good solution, because it only moves the problem to another, even more time-critical part of Flipper. It is not taken in account any further.

Reduce Size of Bytecode

It is also necessary to design the *bytecode* in a way that it *uses only a minimum amount of space*. This concerns the traffic produced by Flippers while exchanging models on the network, where apart from the protocol the bytecode may play a major role. It also concerns the amount of storage space needed for models that are kept locally. The importance of both is again depending on the model of distribution and may also vary in different situations and environments.

Security

The fourth requirement deals with security. It should not be possible for a bytecode to crash Flipper, the underlying system or do unwanted harm in any other form. Of course it is one of the responsibilities of the loader to do integrity checks, but the bytecode may provide special information to make these possible or in some way easier. Note that this touches the efficiency issues. No security checks at all may of course be most efficient, nevertheless they are most important and cannot simply be abandoned. It is obviously important to implement it in an efficient way.
Another form of security is to hide implementation details (e.g. names of variables) in a way that makes it hard to do reverse engineering. This may be especially wanted for model developers that make the functionality of models available to others, but do not also want to provide those with the actual knowledge.
So there has to be support to *secure Flipper* (from invalid or unwanted bytecodes) *and* the *bytecode* itself.

Support Debugging

While transforming the original FML models into the bytecoded form, a lot of information, which is not directly useful for the management task a Flipper performs, is sorted out and does not appear in the bytecode any more. This is of course one reason, why it is possible for a bytecode to be more efficient in several aspects. On the other hand this efficiency may not be as important as the *possibility to debug* a model in a convenient way during the phase of developing. For this, line numbers, original names of identifiers and so on are required in these situations.

Provide Separate Interface for Models

All the requirements that have been listed so far are in one way or another common among bytecodes for different languages and systems. The last one is special to this bytecode, because it deals with an immediate problem of the FML in the context of

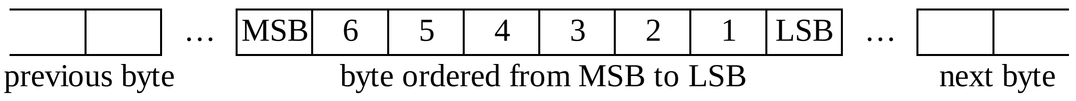
distribution. Section 2.2 of chapter III described that the FML is missing an interface description of models that can be kept separate from the fully implemented model.
As this is important for a distributed Flipper system, the requirement on the bytecode is, to offer this interface-view on models.

2. Overall Decisions

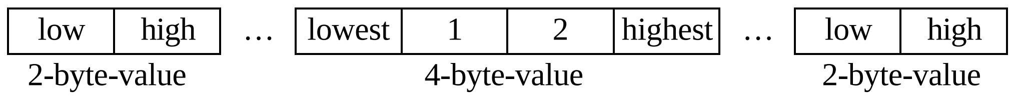
This is the first section to describe internal details of the bytecode. It deals with decisions that affect the bytecode as a hole, not only single parts of its structure, and describes how these help to fulfil the requirements listed previously.

Basic Characteristics

A basic characteristic of each code or data is how the primitive values are represented. Nowadays it is common for most machine architectures to use units of eight bits called one byte as the smallest addressable memory units or at least a multiple of it, with the possibility to manipulate single bytes as well. So it finally is an easy decision to choose these units of eight bits as a basis for the bytecode. Secondly one has to choose, what sometimes is called the sex of a byte, meaning the direction within a stored byte, in which the weight of its bits increases. As for now, Flipper is only available on Windows NT, most commonly used on Intel architectures, the bytecode will use the very same orientation with the most significant bit coming first (to the left, when written in the usual way).



The same reason applies for the choice regarding the order of bytes when more than one is used to represent a value. In this case the byte with the lowest significance comes first (at the lowest address), the others follow in ascending order.



This does not apply to strings. As they are sequences of characters, their components are equally significant in this context and stored in the same order as they appear in the string.
These issues sometimes seem not worth talking about, but the exact definitions are important, because - as described earlier - the bytecode may not only be used on very different architectures in the future, but also be shared among them. Having a fixed way of coding later avoids the need to decode bytecodes according to the architecture they were compiled on.
The same accuracy must be used when defining any part of the bytecode, to guarantee that the decoding of its fields will be possible without misinterpretation.

Characters

The next decision concerns the characters used in the FML. The definition of the language only describes what letters, digits and so on are allowed to form identifiers and what ‘words’ are interpreted as keywords and so on. For the level of bits it only says characters are represented like the type ‘char’ of the C language. This makes them dependent on the system or the compiler respectively.
The bytecode will prescribe to use a subset of the characters used by the current implementation on NT to avoid confusion in a distributed environment: Characters must be represented in the common ASCII standard and extended by one extra bit (set to ‘0’) to a byte. To keep the compatibility to the current system, the range of characters starting from hexadecimal code 0x80 to 0xFF is allowed to be used but not recommended, because it is not standardised. The bytecode will use sequences of the same set of characters to build strings.

Support Security through ...

Additionally some decisions must be made to meet the security requirements.
There are several possibilities to provide the loader with extra information that helps to distinguish correct bytecodes from the unwanted and possibly dangerous ones.
The falsification of a bytecode may happen due to errors on a low level, like while transporting it across a network, because of a corrupted file system or generally due to any hardware error. Basically all these errors should already be detected by the operating system respectively the transport mechanisms.

... Hash Values

As it cannot be simply assumed that the underlying transport layer is completely reliable, one or more fields containing a value that is generated by a so called one-way hash function will be used. This decreases the probability of undetected errors.
Of course some kinds of corruption that destroy basic structures may easily be detected by the loader anyway, because the interpretation of length fields for example may lead to errors. However, because such an implicit test is not enough other corruptions are much more complicated or even not possible to detect.

Even hash values themselves are not a perfect solution. However useful they may be to detect the described errors and simple alterations including intentional attacks on the bytecode, they cannot prevent more sophisticated attackers with a detailed knowledge, who may alter the hash value according to the other changes they made.

There is one theoretical way to solve this problem:
If the loader checks any bytecode in a way one can assure it will not be able to violate the security restrictions, then any attack on a bytecode trying to do so must fail. It must be pointed out that the possible range of checks that can be done in practice is too limited to completely fulfil these expectations. The alteration of certain constants represented in a bytecode, for example, may - indirectly and perhaps solely in combination with other (so far correct) models - lead to severe problems. This can only be detected using additional knowledge and higher level rules, for example combined with a theorem prover. In general this extra knowledge is not available also because it is hard to generate and practically impossible to maintain 'waterproof' against attacks.
Still the loader or alternatively the inference engine can do many important checks, which assure that Flipper itself will not crash etc. In addition to certain modifications to an originally harmless bytecode these checks also detect those constructed by a malfunctioned compiler that may still generate valid hash values. Therefore they finally are very helpful and necessary, but it is hardly possible for the bytecode, to provide direct support for them.

... and Signatures

A second way to keep sophisticated attackers from manipulations may however be very well supported by the bytecode: The use of reliable identifications of the creator of a bytecode like the well-known techniques of electronic signatures. In this case, the hash value would be protected by the same mechanism and therefore guarantee a high probability of detecting any manipulation carried out after the compilation (by a now well known creator).
Of course it would be possible for the signature to directly protect the whole bytecode, not only the hash value. However, this does not lead to significant security improvements, but to the slow process of encrypting the whole bytecode, instead of a much shorter value.
If the creator is unknown, the bytecode may be rejected or only accepted if it can, for example, be proven that it does not directly or indirectly through other models use access modules that perform possibly dangerous actions on files or through the use of SNMP. However, these tests are not simple to implement and neither provided by Flipper at the current state of development nor does anything like a classification of models or access modules exist in the FML.
A field that identifies the creator may additionally be useful to support the possibility of a fast loader, because some tests may be skipped entirely for a trusted creator.

Controlling Versions

There is another related issue. It is about different versions of models and underlying DLLs. The loader must detect models and DLLs that do not match, to ensure that the models are working correctly and again to preserve the integrity of the Flipper system. The first approach that addresses the problem is rather straight forward. Version numbers may be applied to both, the models themselves and to the DLLs. The format of DLLs and how a version number might be applied to them is not the concern of this paper. It does not seem to be a complex problem and therefore it is assumed that it can be done.
Note that the versions of models have nothing to do with the version of the bytecode, which may change as the ongoing evolution of the requirements of the project is reflected.
The version numbers should be split up into two parts, a major and a minor number. A change in the major number indicates a profound change in the model, like a change of names of rules or alteration of parts of the implementation that affect the 'meaning' of rules or objects. In other words, something changed in a way that the model is no longer compatible to older versions. Increasing the later indicates minor bugfixes or at most the addition of rules or objects. These changes may not result in conflicts when the model or DLL is used just like one with a lower minor but same major number.
Version numbers are only useful with the appropriate 'counterparts'. Whenever a model or DLL is used by another model the later must tell which version it expects.
The model loader then must check if all versions fit together, meaning the major version numbers are identical and the minor number of the used model is the same or greater.
Eventually the correctly used numbers offer a means to quickly detect conflicts - even if those were implicit in the purpose (implementation) of a rule or a method in a DLL and so undetectable through checks by the loader, simply because a different version number indicates these changes.

Separated Interface

Version numbers are of course not enough to ensure the wanted security or particularly stability of the Flipper application. The changing of the version numbers may have been forgotten or purposely done in a wrong way. So it is again up to the loader to detect possible conflicts; at least those which may be done without 'background knowledge' like the purpose of rules etc. A test that is possible for the loader is to check the compatibility of the header of rules and methods against their use.
A model 'A', for example, uses a rule '[v1] b1', where v1 is of type 'T1'. The rule is expected to be defined in model 'B', which is imported by 'A'. If 'b1' is unexpectedly defined as '[T2 v2] b1 IF ...' in model 'B' where 'T2' is not related to the 'T1'. Then -

depending on other conditions - the evaluation of ‘b1’ in model ‘A’ will lead to wrong results or in a runtime error of the inference engine.

Problems that are detected at runtime may be more dangerous compared to their detection during the process of loading, because in the first case they might occur during a critical phase of an important problem solving. The system or the administrator is usually more prepared to face these problems at loading time, because they are just one of many other possible errors at that stage. Note that the advantage of doing the checks at load time may diminish when the current strategy to load models manually before their use is changed to a more complex version. The later would at least be useful when Flipper exchanges models in a distributed environment where a dynamic and automated loading “at need” would be of help, but would mix the process of loading into the reasoning.

The test itself is not too sophisticated, because it is more or less only a matching between the headers of rules and their use. If, on the other hand, the information which rules are defined and used in the model that has to be loaded is as hard to gain as it is from the pure FML models, where it is distributed all over the model, then this process becomes relatively costly. During the construction of a bytecode it is possible to extract and store both kinds of information in a common field - the defined rule and method headers of a model as well as the rules and methods it uses. Note that the compiler may already check the use of those defined only in the model itself. These must not appear in the previously described field.

What is left, could be called the imports and exports of the model. The exports are commonly also called the interface. The present definition of the FML does not enjoin or even allow model interfaces. As described in section 2.2 of chapter III, model interfaces would have certain advantages for a distributed Flipper system.

So finally the representation of the information about the exports and imports does not only deal with security issues, but also serves other requirements derived from distributed management.

The decision to meet them at one stroke is to split the bytecode into one part that serves as interface and may be used on its own while the second part carries the rest of the information. This is called the implementation part of the bytecode and is designed in a way it may be shipped, but not used, without the first part. In this way a Flipper that already owns the interface can fetch the implementation from somewhere else as an add-on. The later uses a minimal amount of space because it reuses definitions described in the interface. This is the reason why this approach is preferred over two independent forms of bytecode for the two purposes: knowing about a model as opposed to owning it completely. The only price one has to pay is the necessity to clearly identify the correct matching of the interface and implementation blocks, to avoid the introduction of new security problems.

This can be done by doubling the version information, to make it possible for a Flipper to ask for the matching parts and by providing an additional mechanism. The next paragraphs explain this point and show that it is in fact easy to do.

Note that it makes sense to extend the interface by some fields that actually belong to the imports. The first reason is that the interface may use objects from other models as types for parameters. Their origin is clearly identified in the imports, but should be part of the interface to preserve its independence. The second reason to put even more fields into the interface is to provide a Flipper not owning the full bytecode with extra information, like what other models the whole model (not just the interface) is based on or about its size. This may be needed in the distributed case to decide whether the implementation of a model should be fetched or it might be better to find another Flipper to do the work.

From now on this document will use the following terminology:
The ‘basic imports’ of a model identify all imported models and the objects needed in the headers of rules, methods and in the definition of its own objects, or in other words, for the exports.
The ‘implementation imports’ are all imports of the model but the basic imports.
The ‘interface block’ of the bytecode (the previously described first and stand-alone block) contains all exports of the model plus the basic imports and additional fields to meet some of the requirements listed in the previous section, like information about the implementation which is useful in distributed environments and to provide security. So it is to some extend more than what one would call the interface of a model, but still named after it, because it is its main part.

Reusable Constants

The next overall decision deals with the reduction of the space bytecodes consume. A commonly used method to save space is to find expressions or in this case constants that are used more than once and replace them with a reference to a single representation.
It is obvious that some overhead - regarding the generation time and the extra space contradicting the amount of space saved - must be accepted to achieve this. The methods used during construction and loading of the bytecode are not designed to be compared with compression techniques, but to avoid major and obvious waists of space.
What pieces of the bytecode are likely to occur more than once? There are several expressions in the FML that are represented by strings. Names of models and objects as well as parts of the identifiers and rule headers. Note that the complete headers of rules will be replaced by a different mechanism anyway. Following the definition of

FML strings representing one of the former can never be identical to one of the later, because they must start with different characters.
The bytecode will use a very effective internal method to represent objects, so an object’s name is used only once. Still, objects from different models may have the same name and indeed models themselves are often called just like the object they mainly describe. This is the first case for optimisation.
The next derives from the rules and identifiers, which are syntactically very similar to simple rule headers. For both of them it is common to use identical signatures in the context of different types. Note that in the terminology of Flipper a rule’s ‘signature’ does not include the types of its parameters. It only consists of a list of names that are used to separate the parameters in the FML.
Also represented in the form of strings are the often reused names of DLLs and the functions within them. These must not be present more than once either.
Another possible case derives from the lists of types, which - together with the signatures - form a rule header. They may be common among different rules, but because a very short representation for types will be introduced later the efforts to do the optimisation would outstrip the gain.
Separate parts, described in detail in the next section, for the strings of the names of objects, models and so on, but also for the DLLs will be placed in the bytecode.
The last case deals with constants. It is completely up to the developer of models how often identical constants appear. He may even do the optimisation himself by indirectly introducing a rule for the constant, for example. Still, most constants are going to be collected at one place to make them reusable just like the other things.

References

One particular type of fields - the references - will appear throughout the bytecode. They are not only used to refer to constants, but also to definitions of models, objects, rules and so on.
To make them as small as possible they are not used like byte- or word-aligned addresses into the bytecode, but like counters of the expressions they point to. So a field carrying the value ‘three’ in a context where a reference to a name is expected represents the identifier of the third name defined in the bytecode.
Luckily it is possible to distinguish the different types of references according to the context, so the reuse of values for each of them is not a problem and even helps to keep references small.
Note that those references are used only locally. They do not stand for unique representations of objects or rules or anything else. After a bytecode is loaded, they are no longer relevant and therefore forgotten.

Debug Information

How debug information is integrated along with the standard fields is a non-trivial decision, because the amount of debug information may be huge during the phase of development and not be present at all, or reduced to some basic things, when a model is shipped.
If it is always placed along with the object, rule or any other thing it describes, or alternatively placed at a central place, with a reference residing at the described objects, then the ready developed and shippable bytecode will still contain masses of fields saying “no debug information here”. If on the other hand the normal fields are generally kept free of these references or texts and they are stored separately with references back to the things they describe, then this slows down the process of loading. This is due to the fact that Flipper keeps debug information along with the objects representing parts of the FML.
In the first case on the other hand, the information may be placed into the objects during their “construction”, rather than running through all of them afterwards, attaching the debug fields.
Both approaches are combined in the bytecode. A flag at the beginning of the bytecode tells whether each expression that may need debug information really contains it (or a reference into the debug area) at a specific position or none of them do. Basically this says if the model is created to be debugged or shipped.
Debug fields containing any information may be placed into the debug area in both cases. Even ‘backward references’ as described before are generally allowed, but not defined in this version of the bytecode. The other method is suggested during debugging instead.
The decision, what part of the information is going to be placed directly besides the described expressions and what other parts are references into the debug area is taken individually for each case, but follows the guideline to save space. Even if during debugging it may seem not too important, one must try to prevent exaggerated waste, like putting the same filename a hundred times beside different expressions.
The general solution is, to put everything that may otherwise lead to great waste of space into the debug area. This applies of course mainly to strings. Smaller values like counters may not use more space than the reference itself. They are put directly besides the described expressions.

Order of the Parts

The last overall decision deals with general issues concerning the order of the different parts of the bytecode. The details are implicitly described in the next section. Nearly the whole order is automatically resulting from the following decision: All entries, fields, values and so on that are needed within another entry must be listed before that one. This means in other words, that no so called ‘forward dependencies’ are allowed.

The only way to set up dependencies in the bytecode is through the use of the previously described references.

For the order of the parts of the bytecode this means that all very basic things, like the name constants, which are used from various kinds of places must come first, followed by the most basic FML expressions: the objects. Then, the headers of the imported and exported rules must be listed before the final part, the bodies of the rules.

The reason, why the use of references at a position in the bytecode before the entry they refer to is not wanted, is to allow the loader to read and process a bytecode in one pass. This simplifies its work, because all objects, rules and so on may be generated as soon as they appear in the bytecode. Later, they may be referred to in basically two kinds of situations:

Either, they are used within other expression, like a rule header uses objects as parameter types; Or the expression with the reference adds information to the thing it points to, like exactly one rule body belongs (extends) to one rule header.

In the first case forward references would make immediate checks, whether the referred to object matches, for example, impossible. A second pass to resolve the references, or memory consuming and possibly complex tables of only partly evaluated expressions constructed during the first pass and reduced when possible, would be needed instead.

In the second case, the expression could for a similar reason not be completed immediately. Again, a second pass would be necessary or some expressions would have to be stored until the other one that it referred to will be parsed.

3. The Structure

Now that all requirements and overall decisions have been presented, it is time to describe the structure of the bytecode. This section starts with the general structures and refines them until the whole bytecode is defined in detail. Only the part dealing with the representation of rules will not yet be described at this point, because two different versions will be discussed in the following sections.

The final form of the bytecode in the file will be represented in a BNF-like notation, where each symbol at the lowest level is one of the following:

Symbol:	Representation in the bytecode:
Byte	Unsigned (positive) number not greater than 2^8, represented by 8 bits.
Char	Character in ASCII-representation extended to 8 bits, as described in section 3.2.
Word	Unsigned (positive) number not greater than 2^{64}, represented by four bytes.
ShortW	Unsigned (positive) number not greater than 2^{32}, represented by two bytes.
SmallNumber	Unsigned (positive) number not greater than 32895, where the values from 0 to 127 are represented by one and from 128 to 32895 by two bytes.

These actually non-terminal BNF expressions should be looked at as types. They are not fully described in BNF themselves. All their bits and bytes are represented in the file in the way it is described at the beginning of section 3.2.
To avoid misunderstandings the corresponding type of explicitly stated constants is indicated immediately after each of them, separated by two colons. Numeric constants are noted in hexadecimal form, which is indicated by the prefix ‘0x’.

The ‘SmallNumber’ needs further explanations. It is designed to hold positive values that are expected to be small - or more precisely smaller than 128 - in the usual case, but where the general use of a ‘Byte’ might be too restrictive for special cases.
For the values from zero to 127, it is interpreted just like a ‘Byte’. In this case, the most significant bit is (implicitly) not set.
For values greater or equal to 128, the bit is set and indicates the two-byte form of the ‘SmallNumber’. Its value is calculated, by clearing this bit and shifting the whole byte into the high-byte of a ‘ShortW’. Then the second byte plus 128 (for the already representable values) must be added.
This allows a maximum value of 32895 while only using one byte for the usual cases.

As additional basic expression a string represents a sequence of characters without length field or special character to signal its end:

String ::= (Char)*

In most cases the following expression is used to represent any form of text instead:

Text ::= Length String

‘Length’ is generally used to indicate a number of bytes. It is defined as:

Length ::= SmallNumber

Another expression with the same definition as ‘Length’ is used to represent a number of more complex items or fields in the bytecode:

Number ::= SmallNumber

‘SmallNumber’s are also used to hold the values of most references:

StdRef ::= SmallNumber

The only difference between the last three expressions is their specific meaning. They are only introduced to make the grammar more readable.

3.1 Interface Block

The first part to be described is the interface block. It serves as a coded description of the exports of a model and gives basic information about the implementation part. On the one hand it must be designed as a standalone module and may therefore not contain any links to that other block. On the other hand one must keep in mind that the information stored here will be referred to from within the later.

Header

The block begins with the standard header, which contains the magic number. Such a number is used by many other file formats, mainly in the UNIX world, but also in other systems. Each format is identified by a unique four-byte value. The uniqueness is not guaranteed by a generally approved institution.

IFace_Magic ::= 0xFBC0FACE::Word

The second part of the header informs about the version of the bytecode itself.

```
Version_Major    ::=  ShortW
Version_Minor    ::=  ShortW
VersionNr        ::=  Version_Major  Version_Minor
```

The version numbers are created, used and interpreted in the same way as it has been explained for the versions of models previously.

Additionally it must be stated what variant of bytecode, concerning the debug information, it is. New indicators for different kinds of debug information may be added later. For now only the following are defined:

```
DebugType        ::=  DT_NoInfo | DT_Concluded
DT_NoInfo        ::=  0x00::Byte
DT_Concluded     ::=  0x01::Byte
```

In case of a bytecode marked with ‘DT_NoInfo’ all the optional fields printed in the *[italic]* style are disallowed. Still some debug information may be present in the debug area, as explained later. The fields in *[italic]* must be present for the ‘DT_Concluded’ variant of the bytecode.

Both expressions together define how the bytecode must be interpreted:

```
ByCo_Version     ::=  VersionNr  DebugType
```

The last part of the header identifies the model. It consists of the full name of the model and its version number. Both must be identical to the ones in the corresponding implementation block.
The FML does not suggest how the names of models may be formed to prevent developers all over the world from using identical names for different models. One approach would certainly be to use the same conventions Java uses for its packages, where the internet domain of the developer’s organisation should be used (in reversed order) as a prefix. Another strategy may be, to introduce separate identifications in addition to the name.
Whatever method will be introduced in FML in the future, it must be ensured that the name of the model used in the bytecode is the full name of the model including the identification mechanism.
The version of the model is stated separately.

```
ThisModel_Name ::= Text
ThisModel_ID     ::=  ThisModel_Name  VersionNr
```

All in all the header is defined as:

```
IFace_Header     ::=  IFace_Magic  ByCo_Version  ThisModel_ID
```

Main part

The main part of the interface represents the basic imports and exports of the model and their debug information. It is - like the whole bytecode - designed in a way that all dependencies of an entry are already listed beforehand. In this case it also means that the imports are listed before the exports.
The investigation that analysed the dependencies between models and explained the imports and exports more in detail has already been described in the author's paper about “Consequences of Changes in the Architecture of Flipper’s Compiler”, which is available at the HP Laboratories in Bristol.

Text Constants

As described in the previous section, the names of models and objects are extracted from the rest of the interface block and therefore are reusable.
The first block of (unique) text constants stores these names. Note that one name, the name of the model described by the bytecode, has already been used. It is not listed with the other constants to provide a fast identification of the bytecode, but will be referenced as constant number zero.
The number of constants listed in this part (not counting number zero) is simply followed by a sequence of ‘Text’s:

```
IFace_Names      ::=  Number  ( Text )*
```

The names are going to be referred to, using the following expression:

```
Name_Ref         ::=  StdRef
```

Remember that the ‘StdRef’ is defined as ‘SmallNumber’, which is very space efficient in holding references within the bytecode for conventional models, because they are expected to contain less than 129 name constants.

The second block of constant texts is defined and referred to in the same way. Its entries are counted restarting from zero. They hold the signatures of rules and the names of identifiers of objects:

```
IFace_RuleSigs   ::=  Number  ( Text )*
RuleSig_Ref      ::=  StdRef
```

So the part for all text constants is defined as:

```
IFace_TextCs     ::=  IFace_Names  IFace_RuleSigs
```

Debug Information

This debug information is put in a single list of debug entries and as usual preceded by the number of entries it has.

```
IFace_Debug      ::=  Number  ( Debug_Info )*
```

All entries are going to be referenced using a ‘Debug_Ref’ expression:

```
Debug_Ref        ::=  ShortW
```

The amount of entries is expected to be greater than 128 in many cases, so instead of the more complex reference types applied to other cases, a simple ‘ShortW’ is used. All debug entries are counted starting from one. The value 0x0000::ShortW is used to indicate that no information is present. It may replace any real reference to a debug entry.

Currently, three entries are allowed as debug-information:

```
Debug_Info       ::=  DI_Filename | DI_Description | DI_ThisModel
```

The first two are used in different contexts to store a filename and some other general description:

```
DI_Filename      ::=  DIFilename_ID  DIFilename_Length
                      DIFilename_Value
DIFilename_ID    ::=  0x0001::ShortW
DIFilename_Length  ::=  Length
DIFilename_Value   ::=  String

DI_Description   ::=  DIDescr_ID  DIDescr_Length  DIDescr_Value
DIDescr_ID       ::=  0x0002::ShortW
DIDescr_Length   ::=  Length
DIDescr_Value    ::=  String
```

The third gives debug information about the model itself. It is not referred to from anywhere in the bytecode:

```
DI_ThisModel     ::=  DIThisModel_ID  DIThisModel_Length
                      DIThisModel_Value
DIThisModel_ID   ::=  0x0003::ShortW
DIThisModel_Length ::=  Length
```

```
DIThisModel_Value  ::=  ( Debug_Ref )+
```

To avoid ‘forward references’, this entry must come after the other references it refers to. The first entry in its value must refer to a ‘DI_Filename’ naming the file that contains the FML description of this model. Other fields are optional.

New debug entries created in the future must have the same format as the current: The first two bytes identify the type of the entry. It must not conflict with existing identifiers! The next two bytes state how long (measured in bytes) the body of the entry is. The two values make it possible for a simpler implementation of a Flipper to ignore unused or unknown entries. Any definition is allowed for the value.
Note that the length and value of the first two entries could have been defined as one ‘Text’ expression, without violating the format of the entry. The length has been separated in the grammar to prevent errors in case ‘Text’ is going to change in the future.

The same format of entries will be used for the debug information presented right beside the described expressions. So more than the two IDs from above will be occupied. They may of course as well be ignored by a Flipper.

Basic Imports

There are two kinds of information given in the part of the bytecode for the basic imports. The references to the identifiers of all imported models and those to the objects needed within the headers of the rules and methods, and secondly those in the definitions of new objects within the exports.
As annotated in the previous section, the names of models not directly necessary for the interface are also listed in this part of the bytecode. They have been placed along with the other names to have all of them in a common place. Their presence is compulsory, so the implementation block can rely on the completeness of the information.
The references to the names of the objects are listed immediately after the declaration of the model in which they are expected to be defined in. The whole list, as well as each list of object references, is preceded by a ‘Number’ indicating the number of following entries.

```
IFace_BasicImp  ::=  Number ( Model_ID  [ DI_Model ]
                     ObjectDefList  )*
ObjectDefList   ::=  Number ( Object_Name  [ DI_Object ] )*

Model_ID        ::=  Model_Name  VersionNr
Model_Name      ::=  Name_Ref
Object_Name     ::=  Name_Ref
```

Remember that the *[italic]* parts are mandatory if the debug flag was set to 'DT_Concluded', but not allowed to be present for 'DT_NoInfo', which is used for 'release versions' of bytecoded models. As the definition of the bytecode is complex enough by itself, the definition of the debug fields are listed separately at the end of this section.

The models are going to be referenced by the following expressions. It uses the same mechanism as the references defined to access the constants:

```
        Model_Ref          ::=   StdRef
```

The references to the objects are described in the following part about types.

Types

Before it is possible to describe the export part, expressions for types have to be given. These can either be references to already listed (imported) definitions of objects within the bytecode or general built-in types of the FML.
The definition is quiet vital for the size of the bytecode and its ability to be loaded quickly, because types will be used throughout the code and like references to names, they help to avoid multiple space consuming occurrences of the same objects, which are usual in plain FML.
Types will be used and embedded in different contexts:
Wherever a

- parameter,
- identifier, or in later parts of the bytecode a
- local variable

is used or defined, its type must be given in the bytecode. As mentioned before, it may in most cases either be a built in core type of the FML, like an 'Integer', or an object defined in a model.
At the point of the definition of an object, where its super object is listed, only a non-coreobject is allowed. At the same place, a special indicator representing 'it has no super object' is needed.
When defining the bodies of rules, types must be applied to

- variables and
- constants.

Whenever a type expression represents a built-in type, a constant value will be used. If however it stands for an object, a part of the type expression will be interpreted as a reference to an object definition. Like references to names they are not absolute addresses within the bytecode, but refer to the n-th object definition.
In this way it is possible to design types in a way they use only one byte in most cases.
The following BNF-expressions are used for types:

BuiltInType, Type_Ref, BuiltInTypeC and Type_RefC

The loader will know which of them is meant, because the first byte of each - and this is the only one in most of the cases - is going to use distinct ranges of values.
The most significant bit - if set - indicates that a constant follows. A ‘C’ is appended to the BNF expressions accordingly.
The range reserved for built-in types is fifteen numbers from 0x70 up to 0x7E. For the moment, only these four constants are in use:

```
Type_Char        ::=  0x70::Byte
Type_Integer     ::=  0x71::Byte
Type_Real        ::=  0x72::Byte
Type_String      ::=  0x73::Byte
```

The equivalent fifteen numbers from 0xF0 to 0xFE are reserved for the case with the constant-indicator bit set:

```
Type_CharC       ::=  0xF0::Byte
Type_IntegerC    ::=  0xF1::Byte
Type_RealC       ::=  0xF2::Byte
Type_StringC     ::=  0xF3::Byte
```

This leaves enough capacity for future types. Even if new projects will introduce features like a meta-level, which may require much more built-in types, it is still feasible by adding a second byte, with one (or even more) special constant values for the first byte indicating these cases. For now the expressions are defined as:

```
BuiltInType      ::=  Type_Char       ⌋ Type_Integer
                      ⌋ Type_Real     ⌋ Type_String
BuiltInTypeC     ::=  Type_CharC      ⌋ Type_IntegerC
                      ⌋ Type_RealC    ⌋ Type_StringC
```

To serve the cases where a special condition, like the previously explained ‘has no super object’ is needed, it is enough to declare 0x7F as a disallowed value for the reference and later reuse it as a kind of exception expression. Not to add unnecessary complexity to the loader, the “same” value with the most significant bit set or in other words the value 0xFF is reserved, too.

All together this leaves two ranges of 112 numbers in the byte for references to objects: 0x00 to 0x6F for normal use, which is called ‘Type_Ref’, and 0x80 to 0xEF, which is called ‘Type_RefC’. Both are referring to the same objects, but again the later indicates that a constant is following in the next field.

In each case the highest two values 0x6E and 0x6F (0xEE and 0xEF respectively) are not directly used as a reference. The lower indicates a following 'Byte', the value of which must be added to 110 to get the right reference to the wanted object. The higher number works in the same way, but indicates a following 'ShortW'. The value of the later must be added to 110+256=366. This may not seem to be elegant or fast for the loader in the general case, but given the fact that the typical number of objects used by a model is by far less then 110 it saves a lot of space and still allows a maximum of 65901 objects, for special cases.

'Type_Ref' and 'Type_RefC' are not going to be defined by a BNF expression, but with the aid of a table. As explained before, they are similar to the type 'Byte' (with restrictions on the possible values) in most of the cases and extended by an extra 'Byte' or 'ShortW' in some others. The rows with the values allowed for 'Type_Ref' are marked by a 'T' in the first column, those for 'Type_RefC' by a 'C':

Def.	Values when interpreted as 'Byte' Binary	Hex.	Numbers of the referenced Objects	Type of following extension	Is a constant indicated?	Comment
T	00000000 - 01101101	0x00 - 0x6D	0 - 109	none	no	Usual case
T	01101110	0x6E	110 - 365	Byte	no	The ext. is added to 110
T	01101111	0x6F	366 - 65901	ShortW	no	The ext. is added to 366
	01110000 - 01111110	0x70 - 0x7E	---	---	no	Reserved for 'BuiltInType'
	01111111	0x7F	---	---	---	Reserved / Defined later
C	10000000 - 11101101	0x80 - 0xED	0 - 109	none	yes	Usual case, a constant follows in the next field
C	11101110	0xEE	110 - 365	Byte	yes	The ext. is added to 110
C	11101111	0xEF	366 - 65901	ShortW	yes	The ext. is added to 366
	11110000 - 11111110	0xF0 - 0xFE	---	---	yes	Reserved for 'BuiltInTypeC'
	11111111	0xFF	---	---	---	Reserved / Unused

Exports

The export part contains all the definitions of objects and core objects of the model, including their identifiers. Furthermore it contains the headers of the rules and methods.
Using the previously described references, it is possible to define the export fields referring to the objects already declared in the import and export field. The exported objects are split up into two parts. The first describes the names of the objects, their

parents (or the fact that an object has not got one) and the number of identifiers each of them has.
Like in the FML the identifiers of objects may only subclass those of super objects, but it is not permitted to add new ones. Those unchanged may or may not be listed in the second part of the definition of the objects. For this reason the number of 'Object_Ident's might be zero and does not directly reflect the total number of identifiers the object really has.
The complete number of exported objects is, as usual, stated before the list:

Export_Objects ::= Number (Object_Def *[DI_Object]*)*
Object_Def ::= Object_Name Object_Super Object_IdentNr

The 'Object_Super' describes the super object through a reference to that object. In case the object is an ordinary top-level object or a coreobject a special constant is present instead. Core objects can be distinguished from the others, because additionally the number of their identifiers is zero.

Object_Super ::= Type_Ref | NoSuperObject
NoSuperObject ::= 0x00::Byte
Object_IdentNr ::= Number

The second part of the declaration of the exported objects describes their identifiers. They must be listed separately, because the FML allows to have circular dependencies between objects by using another object as type for an identifier, which by itself (possibly indirectly through other objects) uses the former. A simple way to avoid the unwanted 'forward dependencies' within the bytecode is to (incompletely) declare all objects before the declaration of their identifiers. Then each identifier may be of any type, especially the exported ones, because they have all been listed before.

Export_ObjIdents ::= Number (Type_Ref Object_IdentList)*
Object_IdentList ::= (Object_Ident *[DI_Ident]*)*
Object_Ident ::= Ident_Name (Type_Ref | BuiltInType)
Ident_Name ::= RuleSig_Ref

The identifiers described may later be referenced in most of the cases where references to rules are allowed as well. Within the bytecode they are referenced using the expression:

Ident_Ref ::= StdRef

The last part of the exports declares the headers of rules and methods. It does not distinguish between them:

Export_RuleHeaders ::= Number (Rule_Header *[DI_Rule]*)*

```
Rule_Header        ::=  Rule_Signature  Rule_Arity  Rule_ParTypes
                        Rule_ParFlags  Rule_Options
```

These headers are first of all defined by a rule signature. As long as the representation of a signature is injective there is no further need to split it up into a list. A single ‘Text’ from the constant area is used instead:

```
Rule_Signature     ::=  RuleSig_Ref
```

Secondly the arity of the rule is stated. It represents the number of the parameters following in the next field. It must be greater than zero:

```
Rule_Arity         ::=  Byte
```

The types of the parameters are simply presented one after the other.

```
Rule_ParTypes      ::=  (  ( Type_Ref | BuiltInType )  [ DI_Variable ]  )+
```

The field for the parameter flags consists of two lists of a variable length of flags. One for the bindings, the other one for the functionalities. New lists of flags may be added in the future, but note that this cannot be achieved without creating incompatibilities to older versions of the bytecode. Also note that some of the already defined flags are not used by the current implementation of Flipper. The number of items per list entry is equal to the arity. So it is sufficient to store the number of flags per list only. A special value that is stated right before the list of bindings tells which of the bindings is marked as ‘Persistent’. For this purpose, the flags are counted starting from one. The value zero signals a non-persistent rule:

```
Rule_ParFlags      ::=  Number  PersistentBinding  ( BindFlag )*  Number
                        ( FuncFlag )*
PersistentBinding  ::=  Number
```

The flags are stored in bit fields. Only the ‘value of arity’ lowest bit matter, all other must be set to zero. A bit that is set signals a ‘+’ for a binding flag and a ‘K’ for a functionality-flag. Otherwise it represents a ‘?’ or a ‘U’ respectively. For arities from one to eight, one ‘Byte’ per flag is used. From nine to sixteen a ‘ShortW’, while higher arities result in one ‘Word’ per flag. Note that this limits the arity to a maximum of 32. This is a reasonable limitation, especially because the current Flipper implementation limits it even to sixteen.

```
BindFlag           ::=  Word | ShortW | Byte
FuncFlag           ::=  Word | ShortW | Byte
```

The last value for a rule header defines the options the rule has. Further options may of course be added in the future:

```
Rule_Options      ::=  Byte
```

The options are coded on the level of bits. The value of the byte for the options is computed by taking the logical ‘or’ of values corresponding with the options that are set. The value of the only currently defined option is:

```
ROpt_Overriding::=  0x01::Byte
```

The rules will be referred to in the implementation block to define their bodies:

```
Rule_Ref          ::=  StdRef
```

The summing up of all parts of the exports leads to its definition:

```
IFace_Exports     ::=  Export_Objects  Export_ObjIdents
                       Export_RuleHeaders
```

<u>Additional information</u>

All entries of the next part of the interface block are optional. They exist to give additional information about the implementation block. This can be useful for a Flipper to make decisions about what models it does or does not want to process locally.
Except for the entries defined, this part is constructed identically to the debug area. The entries may serve several purposes, for meta-management or as additional information for distributed environments, for example. It must not be looked at as debug information.
The only entries existing in this part so far are the size of the implementation block, to be able to estimate its memory consumption when loaded, and a short explanation of the purpose of the model, to be possibly displayed in user interfaces:

```
ImplInfo_Size     ::=  IISize_ID  IISize_Length  IISize_Value
IISize_ID         ::=  0x8001::ShortW
IISize_Length     ::=  0x0004::ShortW
IISize_Value      ::=  Word

ImplInfo_Purpose ::= IIPurpose_ID  IIPurpose_Length  IIPurpose_Value
IIPurpose_ID      ::=  0x8002::ShortW
IIPurpose_Length ::= Length
IIPurpose_Value ::=  String
```

The entries follow the same convention as the debug entries. New entries created in the future must follow them as well. The identifiers may overlap with those of the debug area. However, this is avoided for now.

The sequence of entries is preceded by a field telling how many there are. The whole part is defined as:

IFace_ImplInfo	::=	Number (ImplInfo_Entry)*
ImplInfo_Entry	::=	ImplInfo_Size | ImplInfo_Purpose

New entries may simply be added as additional possibility for ‘ImplInfo_Entry’. Even if the BNF grammar by itself allows any orders and repetitions of entries, this may be restricted by the developer of each of them. The previously defined ‘ImplInfo_Size’ for example must not occur more than once, but its position does not matter. It may as well be omitted. There are no such limitations for ‘ImplInfo_Purpose’.

Tail

Finally the tail of this block identifies the organisation that created the interface and the implementation(!). It also contains hash values and a digital signature to secure the bytecode against manipulation.
It is suggested to use a hierarchically structured name for the creator, like the internet domain names to allow an intelligent loader to skip certain tests for efficiency reasons according to the origin of the code. So HP may basically trust all of its own models plus those from a certain division of IBM to which a special contract exists and may therefore only verify the signature and the protected hash value.

Model_Creator	::=	Text

The expressions needed to meet the previously described security requirements are explained in the following paragraphs.
The hash value for the interface uses 128 bit and is constructed using the MD5 message digest function on the part of the interface from the top down to the (not included) tail.
The MD5 algorithm has been chosen because regarding its high standard of security, it is relatively compact to implement. One main advantage over common checksums is that it is said to be computationally impossible - given a text and its hash value - to find a second text with the same hash value.
The algorithm and its predecessor MD4 are precisely described in [RFC1321] and [RFC1320] respectively.
Note that the length of the following sequence of bytes is fixed to 16. This is indicated by the common notation (Expr)$^{*}_{min,max}$ meaning that expression ‘Expr’ must be used at least ‘min’ but no more than ‘max’ times.

HashValue	::=	(Byte)$^{*}_{16,16}$
IFace_Hash	::=	HashValue

The interface additionally contains the hash value of the implementation block. This makes it possible to check if it is really the matching part and has not been corrupted. Finally it makes similar blocks like the hash value, the name of the creator or the signature superfluous in the implementation block.

```
Impl_Hash        ::=  HashValue
```

The method used for the signature to protect the two hash values and to test if the ‘Model_Creator’ field is correct, is not fixed, but choosable by the bytecode builder. For now only two methods are defined:
The first may be used during the phase of development or in secure environments. In fact, in this case the bytecode contains no signature and the method is to always assume that creator and the hash value are correct. So it basically offers no protection at all. It is indicated by:

```
NoSignatureID    ::=  0x00::Byte
```

The second, ‘real’ method uses the common PGP algorithm - explained in [ASLA94] - to generate a signature, using the hash value as its input. The representation of the signature consists of:

```
PGPSignatureID   ::=  0x01::Byte
PGPSignature     ::=  Length  (Byte)*
```

It is recommended that other signatures that may be added in the future will start with a length field. So the administrator may allow an older implementation of a loader to ignore it (knowing this might violate some of the security issues).

Eventually the whole signature is defined as:

```
IFace_Signature  ::=  NoSignatureID  |  ( PGPSignatureID  PGPSignature )
```

All in all the tail of the Interface is defined as:

```
IFace_Tail       ::=  Model_Creator  IFace_Hash  Impl_Hash
                      IFace_Signature
```

The hash value and the signature are placed in the tail of the interface to make it possible for the compiler to calculate the former, while writing the two blocks of the bytecode into a file, and to append the tail afterwards.
Note that the signature cannot be created before the hash value of the implementation block - and therefore the whole block itself - is known.

Summary of the Interface Block

Lets sum up all parts of the interface block of the bytecode:

```
ByCo_IFace      ::=  IFace_Header
                     IFace_TextCs  IFace_Debug
                     IFace_BasicImp  IFace_Exports
                     IFace_ImplInfo
                     IFace_Tail
```

3.2 Implementation Block

The implementation block is designed just as if it was directly appended to the interface block. For example, the rest of the imported objects and the identifiers of objects, which are listed in the first part of this block, are going to be referenced by values starting exactly from where they stopped in the previous part for the basic inputs.
So it cannot be separately generated. This is also true the other way round, because as described earlier, the implementation's hash value is stored in the other block.
This block first of all describes the rest of the dependencies of the implementation part, the imports not already needed by the interface plus the identifications of the DLLs, on which methods and objects may be based.
Then the bodies of the rules - the actual implementation of the model - are represented.

Header

The block starts with a header of the same format as the one of the interface, but it uses a different magic number.

```
Impl_Header     ::=  Impl_Magic  ByCo_Version  ThisModel_ID
Impl_Magic      ::=  0xFBC0C0DE::Word
```

The part that identifies the model must be identical to the corresponding part in the interface block. The version of the bytecode may only differ in its debug type, even tough this may not be useful in many cases.

Text Constants

Like in the previous block, names of objects and signatures of rules and identifiers are extracted from the rest of the bytecode:

```
Impl_Names      ::=  Number  ( Text )*
```

```
Impl_RuleSigs    ::=  Number  ( Text )*
```

The names are going to be referred to as usual, indirectly using a ‘Name_Ref’ or a ‘RuleSig_Ref’ respectively through expressions like ‘Object_Name’ or ‘Ident_Name’.
The first area is additionally used, to hold names of DLLs and functions. These are referenced by:

```
DLL_Name         ::=  Name_Ref
Function_Name    ::=  Name_Ref
```

So this second part for the rest of the constants is defined as:

```
Impl_TextCs      ::=  Impl_Names  Impl_RuleSigs
```

Debug Information

The debug information in this block simply extends the one of the interface block. It is defined just like the later and is allowed to contain the same fields:

```
Impl_Debug       ::=  Number  ( Debug_Info )*
```

Imports

What exactly is represented in the imports and also how is mainly influenced by two things. The information about other models that may be taken into account and what is finally needed to enable the loader to do the appropriate tests described in previous sections.
In the ‘minimal’ case, the available information about other models is reduced to the interfaces of the imported models. A developer of a model - and therefore also the compiler in use - does not need to know more than that, but can of course not do a good job when less information is provided. It is important to see that this information is sufficient to write a new model based on others, without generating unsolvable dependencies. In fact, the bytecode interface was designed to serve this purpose. On the other hand it would not be acceptable to simply copy all the definitions of the interfaces of the imported models into the bytecode, for the tests the loader later may make. They must be reduced in a way the loader has to do fewer checks, but the security and other requirements are still met sufficiently.
A maximum reduction to the exact names and versions of the imported models, which is already stored in this model’s interface anyway, would be sufficient, if creators of models (the developers and the compilers) could be trusted regarding their attitude and capability.
As this is not always the case, it must be testable, whether the definitions expected by a model (its imports) conflict with those of the models already loaded. The most spe-

cific information, by which it is still possible to guarantee the existence of such a test, is of course to list the exact needs of the model instead of the definitions of the imports.
However, regarding the ability to efficiently do the tests, a mixture of both is used for the bytecode. It is not called a compromise, because it is faster to test than to compare a complete list of imported definitions and does not consume more space in the bytecode than the most specific information would need. It is even faster to test and easier to implement than the later:
Model ‘A’, for example, defines two objects: ‘A1’ and ‘A1Sub’, which is derived from the former. It also defines a rule: ‘[A1 x] exists IF …’. A second model ‘B’ imports only model ‘A’ and uses the rule only with variables of type [A1Sub].
The most specific specification of model ‘B’s imports - regarding only the rule -would be to say it uses a rule with the header ‘[A1Sub x] exists’. But because the compiler - like the developer - knows about the interface of ‘A’ it can as well say the rule ‘[A1 x] exists’ is used and therefore prevent the loader from a search whether ‘A1Sub’ has a super object matching the rule.
In case of a second object derived from ‘A1’ that is additionally used in model ‘B’s calls to the rule, the most specific information also consumes more space, because both possible calls would have to be listed, instead of one ‘[A1 x] exists’.
On the other hand, the chosen approach is more restrictive on future changes of model ‘A’, but it is still less restrictive than the recommendations for the common basis of models with the same major version number and therefore acceptable.

At the beginning of the imports, the not yet given definitions of objects are matched with a reference to the model they are defined in. This is followed by a list of identifiers assigned to their objects. Like before, in the exports, this helps to avoid ‘forward-dependencies’ from types of identifiers to other objects.
The names of the identifiers and objects are represented by references into the constant areas.

```
Import_Objects   ::= Number ( Model_Ref   [ DI_Model ]
                     ObjectDefList )*

Import_ObjIdents ::= Number ( Type_Ref  Object_IdentList )*
```

Note that no super object of the imported objects needs to be named.
The list of identifiers reuses an expression defined in the interface exports. The debug information for the objects and identifiers is also defined within these.

The next part of the imports enlists the rules matched with the models they come from. The signatures are taken from the constant area. Each is followed by the arity and lists of the types and flags. It is defined very similar to ‘Rule_Header’ in the exports of the interface, but has no option list:

Import_Rules ::= Number (Model_Ref ImpRule_List)*

ImpRule_List ::= Number (ImpRule_Header *[DI_Rule]*)*

ImpRule_Header ::= Rule_Signature Rule_Arity Rule_ParTypes Rule_ParFlags

So the two parts together define the whole part for the imports:

Impl_Imports ::= Import_Objects Import_ObjIdents Import_Rules

An Extension for the Future

After the imports would be the right place to list the rest of the definitions of rule headers and objects, which the developer of a model does not want to make public. A mechanism to declare private or explicitly public rules does not yet exist, but might be useful to have.
The definition of this part would be identical to the interface's exports. No other fields must be altered additionally.

DLLs

All DLLs the model is based on, are listed, using a reference to the name and a version number, in this part of the implementation block. Each of them may be listed several times, under a different category. These categories exist to enable even low level models to work on different systems, using different DLLs.
The first number counts the categories, the second is for the numbers of DLLs, which is equal for all categories:

Impl_DLLs ::= Number Number (Category_Name DLL_List)*

A category, which is basically representing a system, is identified by a 'Text'. No reference is used, because it may not occur more than once anyway.

Category_Name ::= Text

DLL_List ::= (DLL_ID)*

DLL_ID ::= DLL_Name VersionNr

The equivalent DLLs for different systems must be placed at the same position within each of the lists. This assures that the same value for a reference to a DLL always addresses the right one. Only the loader knows which system Flipper is running on, so it is able to choose the right category.
References to DLLs are defined like the other references, where more than 128 references are unlikely to be needed:

```
DLL_Ref          ::=  StdRef
```

Constants

It has been reported previously, how the types of constants are indicated, using the 'BuiltInTypeC' for constants of built in types and Type_RefC for constants of non-core objects or at least parts of them. This serves the context expression of the FML that is declared obsolete in the specification, but still in use.
The constant area is devided into two parts. The first holds a sequence of 'Word's, used for the FML types 'integer' and 'real'. The definition of the former is not restricted to 32 bit in the FML, but the Flipper implementation does. As this seems to be a reasonable limitation, it is used for constants of 'integer' in the bytecode as well. The 32 bits are interpreted as the usual signed two's-complement integers.
FML values of type 'real' are said to be the equivalent of the "C float" type on the architecture. As most commonly the IEEE 754 format for floating point numbers with single precision is used, the bytecode prescribes the use of this format with the exception of the large number of distinct NaN (Not-a-Number) values, which are not treated differently on most of the architectures anyway. All values from 0x7F800001 to 0x7FFFFFFF and from 0xFF800001 to 0xFFFFFFFF must be treated as NaN. This exact predefinition serves the required independence from the platform.
Note that both, 'integer' and 'real' are actually represented in a 'Word' and must be reinterpreted by the loader. This makes it possible to treat a field in both ways, which is not likely to occur, but saves space if it does, although it does not increase the complexity of the loader.

The first constant area is defined as:

```
Impl_WordCs      ::=  Number ( Word )*
```

To access its content, two expressions are used. Each of them represents the type, how the loader will interpret the addressed field:

```
Const_Int        ::=  Constant_Ref
Const_Real       ::=  Constant_Ref
```

Where the reference is defined as usual:

```
Constant_Ref     ::=  StdRef
```

It does not make sense for constants of type 'Char' to be kept in the constant area, because they do not take more space as references. To have all BNF expressions for constants in one place it is also defined here:

```
Const_Char       ::=  Char
```

The second area for constants consists of two sequences of fields with a variable length. The first is used to represent constants of the FML type ‘string’, the second is used for the constants of non-coreobjects:

```
Impl_VarLenCs   ::=  Impl_TextCs  Impl_ObjectCs
Impl_TextCs     ::=  Number  ( Text )*
Impl_ObjectCs   ::=  Number  ( ObjectCField )*
```

The constant FML-strings are stored in a list of ‘Text’s. The constant non-core objects are represented by ‘ObjectCField’s. Each of them defines all or just some identifiers of one object. An ‘ObjectCField’ first of all contains the number of identifiers it describes. This is followed by a list of pairs, of a reference to an identifier of the object and the constant value for that identifier.

```
ObjectCField    ::=  Number  ( Ident_Ref  Const_Any )*
Const_Any       ::=  Const_Char  |  Const_Int  |  Const_Real
                     |  Const_String  |  Const_Object
```

The kind of constant that has to be chosen is defined by the type of the identifier. The compiler must take care not to create ‘forward dependencies’ at this point.

Each of the two kinds of variable length constants are referred to by a new range of values represented by the following expressions:

```
Const_Object    ::=  Constant_Ref
Const_String    ::=  Constant_Ref
```

So the whole part for constants is left to be defined as:

```
Impl_Constants  ::=  Impl_WordCs  Impl_VarLenCs
```

Final Implementations

The most complex part of the bytecode finally describes how rules and methods are implemented. It is based on many things already defined somewhere in the previous parts of the implementation block and on those from the interface.

The first part of it matches the coreobjects of this model with the DLLs and gives references to the names of the functions for hashing, printing and the test for equality, in exactly that order:

```
Impl_CoreObjectList ::=   Number  ( Impl_CoreObject )*
```

```
Impl_CoreObject ::=  Type_Ref  DLL_Ref  Function_Name
                     Function_Name  Function_Name
```

The second part lists the DLLs and functions for the methods. This is the first and only place where it is stated which of the previously defined rule headers actually belongs to a method:

```
Impl_MethodBodyList    ::=    Number  ( Impl_MethodBody )*
Impl_MethodBody    ::=   Rule_Ref  DLL_Ref  Function_Name
```

The last part is represented by the 'Impl_RuleBodyList' expression, which finally describes the bodies of the rules. This part is described separately in the next section, because of its complexity.

The three parts together are represented by:

```
Impl_Impl          ::=  Impl_CoreObjectList  Impl_MethodBodyList
                        Impl_RuleBodyList
```

Summary of the Interface Block

Lets sum up all parts of the interface block of the bytecode:

```
ByCo_Impl          ::=  Impl_Header
                        Impl_TextCs
                        Impl_Debug
                        Impl_Imports
                        Impl_DLLs
                        Impl_Constants
                        Impl_Impl
```

Collection of the Debug Entries

The debug entries that are not listed in the debug area are described in the following paragraphs.
Some expressions are defined beforehand, to make the debug entries more readable. The first entry is used to describe the original position of an expression within a file. It consists of the filename and the line number where the definition (comments possibly included) starts and where it ends:

```
Debug_Position  ::=  Debug_Filename  Debug_Line  Debug_Line
Debug_Filename ::=  Debug_Ref
Debug_Line       ::=  ShortW
Debug_Descr      ::=  Debug_Ref
```

The debug entries not appearing in the debug area are defined as follows:

‘DI_NoInfo’ is allowed for every other ‘DI_’ expression. It tells that no debug information is present:

```
DI_NoInfo        ::=  0x0081::ShortW  0x0000::ShortW
```

‘DI_Model’ lists the name of the FML file where it is defined in, plus an optional description. The length field must be chosen accordingly:

```
DI_Model         ::=  DI_NoInfo
                      | ( 0x0082::ShortW
                          ( 0x0002::ShortW | 0x0004::ShortW )
                          Debug_Filename [ Debug_Descr ] )
```

The definition of Objects is either described by its position in a file or by a textual description entry for definitions:

```
DI_Object        ::=  DI_NoInfo | DI_Position | DI_Definition
DI_Position      ::=  ( 0x0083::ShortW  0x0006::ShortW
                      Debug_Position )
DI_Definition    ::=  ( 0x0084::ShortW  0x0002::ShortW  Debug_Descr )
```

Especially during the lists of the exports, the same filename will be referred to over and over again. This is not superfluous, because a later version of the FML may allow a model to be distributed over several files.

The debug information for the objects' identifiers and rules is just the same:

```
DI_Ident         ::=  DI_NoInfo | DI_Position | DI_Definition
DI_Rule          ::=  DI_NoInfo | DI_Position | DI_Definition
```

For variables no ‘DI_Position’ is allowed:

```
DI_Variable      ::=  DI_NoInfo | DI_Definition
```

3.3 Bodies of Rules

FML objects are abstractions of objects from the real world and models form a kind of a wrapper around them and the rules related to them, in one sense or another. Rules, on the other hand, describe the actual connections and relations between those objects. They show how and to what extent objects from in the real world affect each other.

A second class of rules, the action rules, perform or trigger these affections by themselves to fix a problem or to change some kind of state. Rule headers give certain blocks of descriptions, or actions respectively, a means of being accessed and reused several times from elsewhere. The blocks themselves are called the rule bodies.
The bytecode representation of everything around these bodies has already been described. It forms a large part of the grammar of the bytecode, just like the equivalent expressions in the FML form a large part of the grammar of the language. In opposition to this, the largest part of an average model consists of the bodies of rules. For this reason, a compact representation of rules is vital for space efficient bytecodes.

Declarativeness in the Bytecode

This section describes, how the bodies of rules may be represented within the bytecode without loosing the advantages of the declarativeness.
The main advantage of the declarative-like representation is the same, as it is for declarative languages in general. As these only describe things, but do not say how this description must be proceeded or how a result may be calculated, they leave this choice to the interpreter.
Especially in the context of modelled environments, different 'how's may even serve different purposes, but all be based on the same description. If a bytecode similar to a machinecode (but on a higher level of abstraction) would have been chosen, the algorithms - now embedded in the inference engine - to gather the wanted information from the trees of rules and sub-expressions, would be included in the bytecode.
The inference engine already uses different kinds of algorithms for the prove to find one or all solutions and for the recently added diagnostic feature. Keeping the models separate from these algorithms makes it easy to improve algorithms or to add new ones and so on. If they were coded in each bytecode, all of them would have to be rebuilt, to be updated to the newest version.
Additionally, it is very inefficient concerning space to store the algorithms over and over again in each of the bytecoded models. Also, assuming the algorithms were stored implicitly in the expressions forming a rule body (in the bytecode instructions generated for them), then this clearly takes a lot more space, than declaring the existence of such an expression only, as it is done right now.
Most of the advantages apply for the declarative parts of the FML, but hardly for the action rules. Still, the kind of representation chosen for rules within the bytecode allows a seamless integration of these non-declarative parts.

Kinds of Rule Bodies

The first value of all expressions used in rule bodies is a one 'Byte' expression identification (called 'EID_*'). The individual values are chosen in a way that similar expres-

sion may be detected all together by simple logical operations. They are summarised further down.
The body of a pure declarative query rule is defined by its identification plus an expression for the body:

```
Rule_IfBody       ::=  EID_IfBody  Body_Expression
```

Action rules are expressed in a similar way. Like the 'WHERE ... DO ...' expression of the FML, they posses two sub-expressions: one for the condition and a second for the action:

```
Rule_WhereDoBody ::=   EID_WhereDoBody  Body_Expression
                       Body_Expression
```

No other types of rules are allowed for now:

```
Rule_BodyExpr  ::=  Rule_IfBody  |  Rule_WhereDoBody
```

Note, that the grammar of bytecode does not distinguish between query and action expressions. In the FML, the former, which are a subset of the later, are the only expressions allowed within queries, like in the whole 'IF ...'-body and in the condition expression of 'WHERE ... DO ...'. The same is of course valid within the bytecode, but as a simplification the difference is not expressed in the grammar, but as a semantic condition. The 'Body_Expression's are defined further down.

Variables

Variables are used to temporarily hold object instances and to pass these between the different uses of rules in a rule's body. They have a special status, because they are the only FML elements with a fixed and very limited scope. All objects and rules may be used throughout the model and in all other models importing the former.
Usually, variables appear more than once within a body, what once again offers the opportunity to optimise by extracting them to a definition block (per body).
Each rule body gets a list of variable declarations. These may either be real variables, represented by a 'Type_Ref' or a 'BuiltInType', or a constant, expressed by a 'Type_RefC' or a 'BuiltInTypeC' with a following 'Const_Any', which was previously defined as a 'Char' or a reference into one of the constant areas:

```
Rule_VarDeclList ::= Number  ( Variable_Decl  [ DI_Variable ]  )*
Variable_Decl     ::=  ( Type_Ref  |  BuiltInType )
                       |  (  ( Type_RefC   |  BuiltInTypeC )  Const_Any  )
```

Not all variables are listed here. The type of some variables has implicitly been stated in the header of the rule, where the parameters were defined. So the list only contains variables declared in the body.

Note, that neither the parameters nor the listed variables have a name. Just like other expressions in the bytecode, they are referred to by a value. So the third variable declared matches the reference with the value ‘two’. The value of the reference is lower by one, because the counting of the variable declarations, beginning with the first parameter, starts with the value zero.
The reference is not called ‘*_Ref’ as usual to indicate the different, local scope. For each rule, the references restart to count from zero:

```
Variable_Index    ::=  StdRef
```

Expressions Forming the Bodies

As said before, all expressions start with an identification, which allows the loader to branch into the right code while it reads the bytecode, even without the need to analyse the whole expression first.
For the boolean constants, this identifier is identical to the whole expression:

```
Expr_True         ::=  EID_True
Expr_False        ::=  EID_False
```

The other three final expressions have an additional field describing the use of a relation:

```
Expr_RelUse       ::=  EID_RelUse  RelationUse
Expr_Retract      ::=  EID_Retract  RelationUse
Expr_Assert       ::=  EID_Assert  RelationUse
```

The later two are only allowed in action rules.

The ‘RelationUse’ refers to a rule signature, which is identical to the signature of the relations used, and contains the arity. This is followed by ‘arity’ numbers of variable indices, which refer to the variables passed as arguments to the used relation. Note that no references to imported or other exported rules are listed. The inference engine finds those rules at runtime according to this relation and the actual types of the variables at that moment.
The ‘RelationUse’ is not defined as a ‘Body_Expression’ on its own, therefore it has no identification:

```
RelationUse       ::=  Rule_Signature  Rule_Arity  ( Variable_Index )*
```

All expressions described so far are call ‘final’, because they do not split up into any other body expressions. The following, in contrast, make use of other expressions.
The simplest of them represents the FML ‘~’-symbol, signalling the negation. It has only one sub-expression and does therefore not really branch the body:

```
Expr_Not         ::=  EID_Not  Body_Expression
```

The and- and the or-expressions are - to some extend - exceptions, because they do not have a constant number of sub-expressions. Therefore it is listed explicitly:

```
Expr_And         ::=  EID_And  Number  ( Body_Expression )*
Expr_Or          ::=  EID_Or  Number  ( Body_Expression )*
```

The last three expressions are allowed in action rules only. Two different expressions are used, to represent the ‘IF ... THEN ... [ELSE ...]’-clause of the FML. Only one of them includes the sub-expression for the ‘ELSE’-branch. In both cases, the first sub-expression is used for the condition and must not contain action (in other words non-query) expressions:

```
Expr_If          ::=  EID_If  Body_Expression  Body_Expression
Expr_IfElse      ::=  EID_IfElse  Body_Expression  Body_Expression
                      Body_Expression
```

Finally, the FML-‘FOR ... DO ...’ expression is represented in a similar way. The first sub-expression contains the query and may again not contain action expressions:

```
Expr_ForDo       ::=  EID_ForDo  Body_Expression  Body_Expression
```

The ‘Body_Expression’ collects all possible expressions allowed in bodies. As said earlier, the bytecode grammar does not separate expressions allowed in actions only:

```
Body_Expression::=  Expr_True | Expr_False
                    | Expr_RelUse
                    | Expr_Retract | Expr_Assert
                    | Expr_Not
                    | Expr_And | Expr_Or
                    | Expr_If | Expr_IfElse
                    | Expr_ForDo
```

<u>Expression Identifications</u>

The values of the expression-identifications are:

```
EID_IfBody       ::=  0xC0::Byte
EID_WhereDoBody  ::=  0xC1::Byte


EID_True         ::=  0x01::Byte
EID_False        ::=  0x00::Byte
```

EID_RelUse	::=	0x08::Byte
EID_Retract	::=	0x02::Byte
EID_Assert	::=	0x03::Byte
EID_Not	::=	0x80::Byte
EID_And	::=	0x84::Byte
EID_Or	::=	0x85::Byte
EID_If	::=	0x88::Byte
EID_IfElse	::=	0x89::Byte
EID_ForDo	::=	0x8A::Byte

They all follow the following scheme, which may be used by the loader to quickly identify certain groups of identifications.

¬final	body	0	0	act	multi	c_1	c_2
MSB	6	5	4	3	2	1	LSB

The most significant bit is set if the expression is not final. Bit 6 is set for complete bodies (‘Rule_IfBody’ and ‘Rule_WhereDoBody’). These can never be final, so it may only be set, if the MSB is set, too.
Bits 5 and 4 are unused and set to ‘0’ in any case.
Bit 3 is set for expressions not allowed in query rules, while bit 2 a special kind of expressions, for the case in which bit 4 is not set: the ‘multi’ expressions ‘Expr_And’ and ‘Expr_Or’.
The two least significant bits are used to differentiate expressions with identical characteristic.

<u>Complete Rule Bodies</u>

All rule bodies of the model are packed in one list, which is - as usual - preceded by the number of entries:

Impl_RuleBodyList ::= Number (Impl_RuleBody)*

Each declaration of a rule body consists of a reference to the exported rules, the declaration of which it completes, of the list of variables and finally the expressions forming the structure of body:

Impl_RuleBody ::= Rule_Ref Rule_VarDeclList Rule_BodyExpr

CHAPTER VI - IMPLEMENTATION

The final chapter describes the implementation of the loader and builder for the byte-code and their integration into the existing Flipper application.
Its main purpose is to present the overall concepts and to give an understanding of the ideas behind the especial ways the implementation is realised. This is especially important, because the connections among different blocks of written program code are usually hard to see at the first glance. On the other hand, the code is well commented, so there is no need to describe the details at a lower level.

1. Basic Concepts

The whole architecture of the implementation that integrates the previously described bytecode into Flipper is designed to fulfil the requirements derived from scenarios that are explained in this section. They origin in the recombination of different roles of Flipper.

These roles are:

- to generate bytecodes from FML models,
- to load bytecodes for use,
- to perform management tasks and
- to test (debug) models.

One extra role still exists within Flipper from the time where no bytecodes existed:

- to directly load FML models for use.

Note that this way to split the Flipper application into different roles is very rough, but detailed enough to serve its purpose.
The first and the very last role, for example, contain all those things like syntactic and semantic tests the FML models.
"To perform management tasks" covers most of the technology Flipper stands for, including the different "user-to-Flipper" and "Flipper-to-Flipper" interfaces as well as the inference algorithms. The "test/debug" role contains the same things (used for a different aim) plus some features that are not needed for normal management like especial debugging features.

The basic thought is that the roles a Flipper with integrated bytecode features plays may be devided into two groups: Those needed for management tasks and those to develop new models and to service old ones.

The roles needed for a “Flipper management application” would only be:

- to load bytecodes for use and
- to perform management tasks

From the point of view of this paper the typical scenarios for this Flipper involves the following parts of Flipper’s code:

$$\begin{tikzcd}
\text{Bytecode} \arrow[r, "Loader,\ checks"] & \text{PreStore} \arrow[r, "commit"] & \text{Flipper Stores} \arrow[d, Leftrightarrow] \\
& & \text{Management Tasks}
\end{tikzcd}$$

While performing some checks, the loader fills the PreStore with the data from the bytecode. After success it is transmitted into Flipper’s stores via the commit code where it is used for management tasks. The role and location of the PreStore will later be looked at more closely.
As described further up, the splitting of roles leads to the wanted smaller Flipper code for the management tasks, because no full FML parser is needed there.

Taking the role “to generate bytecodes from FML models” only, leads to a “standalone FML compiler”. The following scenario would apply for this case:

$$\begin{tikzcd}
\text{FML-Model} \arrow[r, "Parser,\ checks"] & \text{PreStore} \arrow[r, "Builder"] & \text{Bytecode}
\end{tikzcd}$$

It represents the process of creating bytecodes from FML models.

There are two possibilities where to put the “test/debug” role. At a first step it could simply extend the management Flipper. This corresponds to the current Flipper implementation.

$$\begin{tikzcd}
\text{Bytecode} \arrow[r, "Loader,\ checks"] & \text{PreStore} \arrow[r, "commit"] & \text{Flipper Stores} \arrow[d, Leftrightarrow] \\
& & \text{Debugging}
\end{tikzcd}$$

The scenarios is nearly equivalent to the management scenario, especially because the tasks that are performed on the Flipper Stores are hardly of interest in this section. Still the loader might be different in both cases, because the loader for the pure “management-Flipper” must usually not deal with bytecodes carrying debug information.

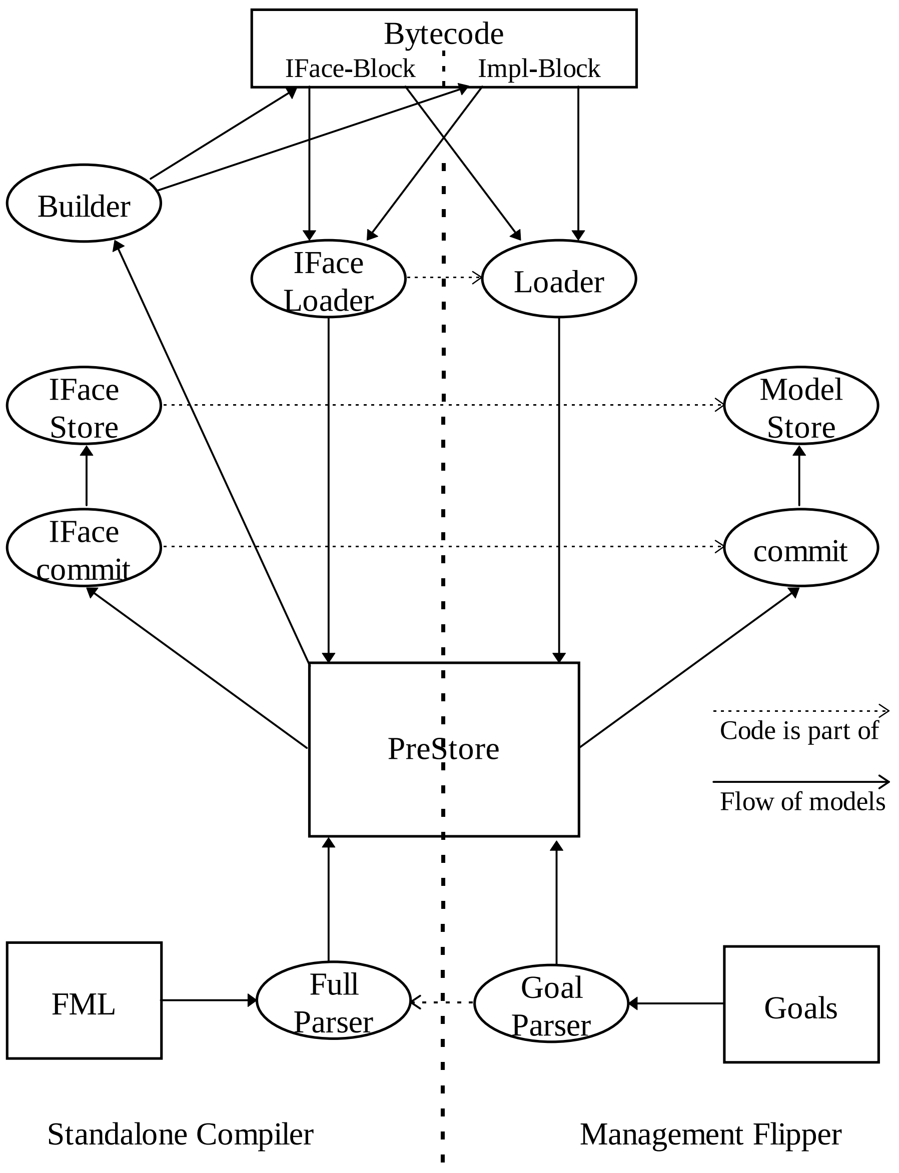
Alternatively one could combine the roles into a “Flipper model development kit”, where mainly the same scenarios exist:

$$\begin{tikzcd}
\text{Bytecode} \arrow[r, "{\mathit{Loader},\ \mathit{checks}}"] & \text{PreStore} \arrow[r, "\mathit{commit}"] & \text{Flipper Stores} \arrow[d, Leftrightarrow] \\
 & & \text{Debugging} \\
\text{FML-Model} \arrow[r, "{\mathit{Parser},\ \mathit{checks}}"] & \text{PreStore} \arrow[r, "\mathit{Builder}"] \arrow[uur, dotted] & \text{Bytecode}
\end{tikzcd}$$

The dotted line stands for a further possible scenario, of directly loading FML models into Flipper, which was the usual scenario before the integration of the bytecode. It might still be useful in certain phases of the development of models to save turn-around times compared to the generation plus the reloading of bytecodes.

The requirements of the scenarios lead (influenced by the existing architecture) to the following image that describes the:

- flow of model elements (full lines) like pure FML, bytecodes and goals; and the
- relationships of components in terms of containment (dotted lines).



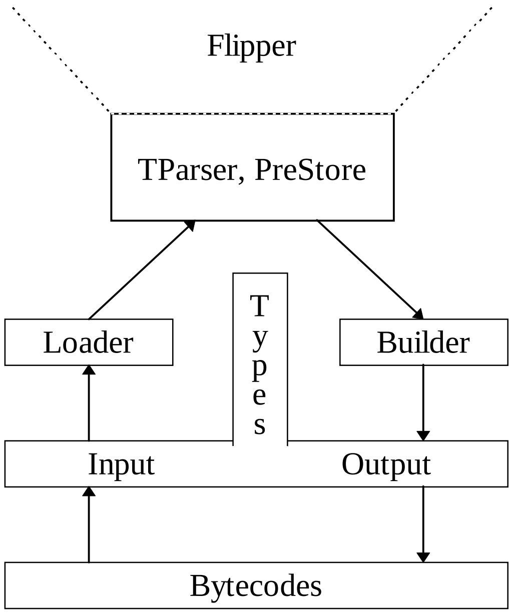
Model flow in two Flipper variants and the relationships of their components

The management Flipper only needs a small parser to process goals, which is a subset of the already existing parser plus an additional loader for the bytecodes.
The standalone compiler does not need a full model store. A store for the interfaces of other models is enough to perform tests when generating bytecodes. The parser is about the same as the already existing one. A loader for bytecode interfaces to fill the interface store and a full bytecode builder are also necessary.

Neither of the sides corresponds with the existing Flipper application, even when not taking the loader and builder into account. Even worse “PreStore”, “full Parser” and “commit” are located within the same C++ class “TParser” and are not very easy to separate.

For the implementation of the bytecode this means that only the missing parts needed to make the bytecode “work” are added, but at least in a way that does not lead to further complications for a future splitting of the current Flipper.
This is achieved by subclassing the full loader from the loader for the interface. As described earlier, bytecodes should only be generated completely. Following this, the bytecode builder will not be split up into several classes.

As the ‘TParser’ class is the only point responsible for the loading and processing of models, both loader and builder will be instanced and started from there.
For the implementation loader and builder are not as clearly separated as it may seem in the diagram above. They are - of course - based on the same grammar, which must somehow be reflected in the C++ code.
The idea here is, to extract those parts from the grammar that may be interpreted independently from their context. The type ‘String’ for example cannot be read from a bytecode without the information of the length. ‘Text’ on the other hand is an independent expression because it contains the length field.
All those expressions of the bytecode grammar are mirrored as C++ types and get separate input and output functions.

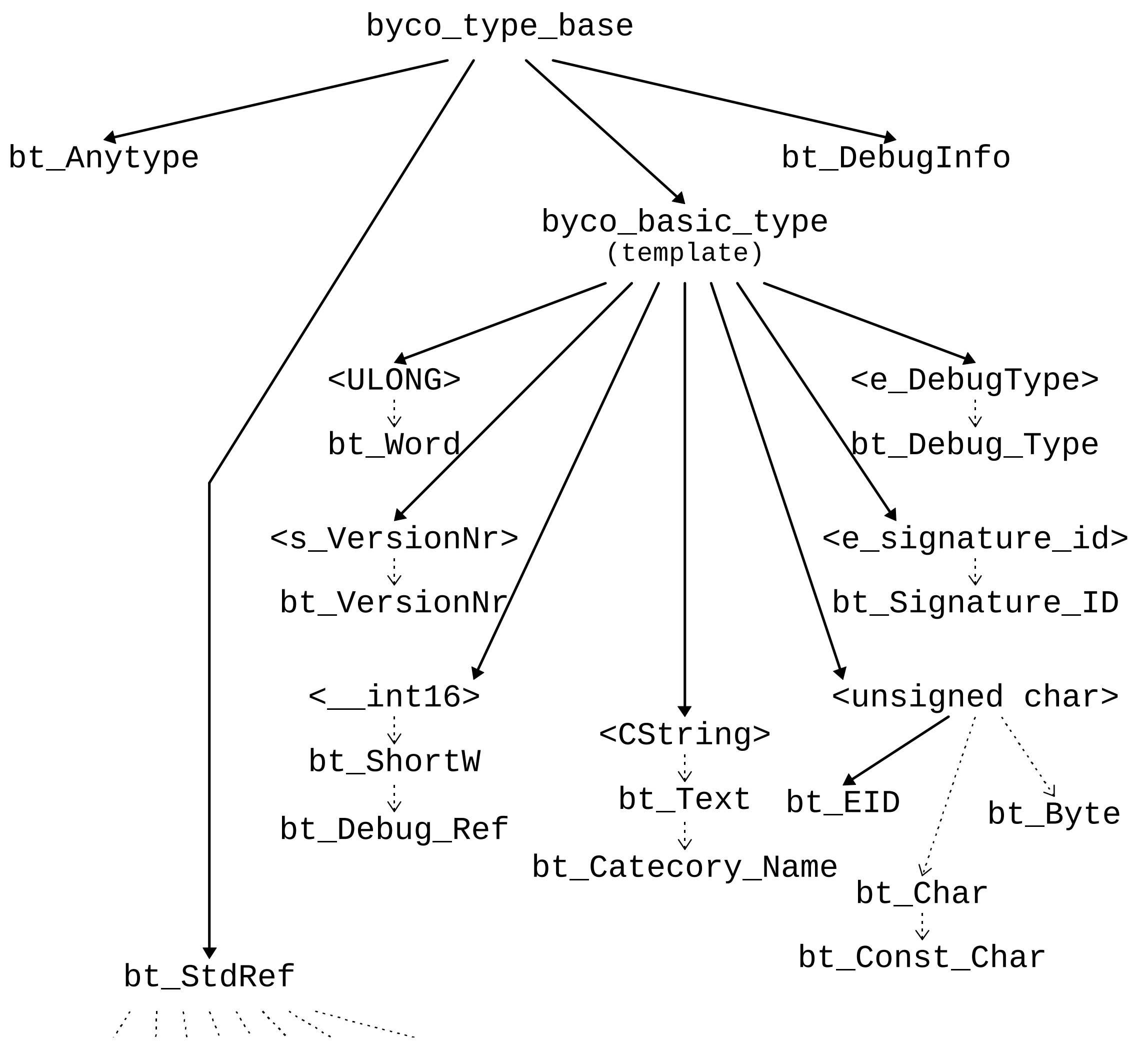


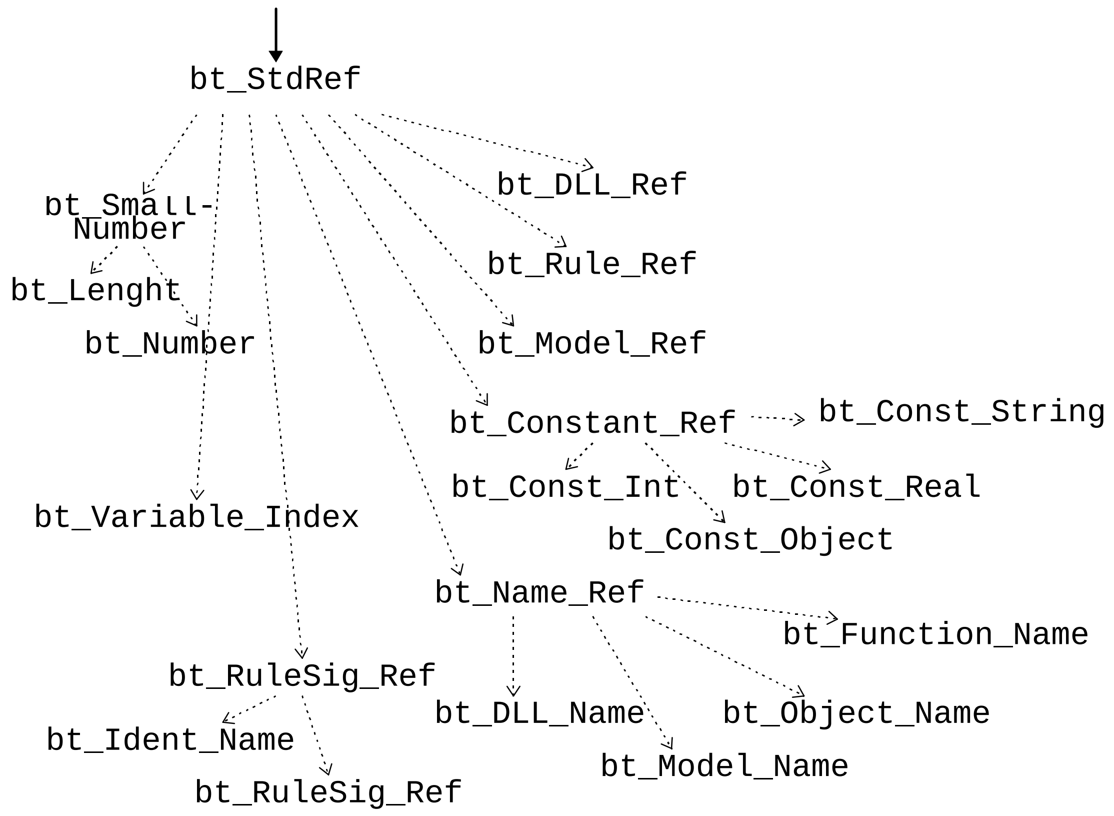
Attachment of the components for the bytecode

2. Implementation Details

2.1 Mapping between the Bytecode Grammar and C++ Structures

The lowest level of the grammar, the basic types and some expressions on the next level are directly mapped to C++ types. These are either subclasses from the root-type "class byco_type_base" or "typedef"s for them.
For elementary types, like 'Word', 'Char' and some simple expressions like 'VersionNr' an intermediate class template "byco_basictype" is used.
The following diagram shows all classes and type definitions in a hierarchical way:





C++ types reflecting parts of the bytecode grammar

Unlike the other classes in Flipper, all names of the classes for types of the bytecode do not start with a capital letter, but with the prefix "bt_" for "bytecode type".
The class "bt_AnyType" encloses the expressions 'Type_Ref' and 'BuiltInType'.

The separation of the types from the loader and builder inclusive all type synonyms has the advantage that specific types may be changed without the big effort of changing code that is distributed over many files.

The definitions and the code for these types can be found in "ByCoTypes.h" respectively "ByCoTypes.cpp"
For all classes - and therefore automatically for the derived type definitions - separate global operators "operator >>" and "operator <<" for input and output from and to C++ streams are defined. They are located in the file "ByCoTypesIO.cpp".

The more complex expressions of the bytecode grammar are mapped to member-functions of the builder and loader. No extra functions exist for simple (but context sensitive) expressions that occur only once.

2.2 Bytecode Builder

The whole builder is packed into “`class ByCoBuilder`”. It is used simply by instancing the class with a reference to the PreStore and later calling the “`build()`” function. This generates the two bytecode files, the interface and the implementation block, by directly accessing the data from the PreStore. The files have the same name, but a different extension. The name is equal to the model name or set to the value of an optional parameter for the name during instancing.

Internally, the process of building is distributed among non-public member functions. They are named after the different expressions forming the bytecode on the highest level of abstraction (below the separation into the two blocks) and carry the prefix “`build_`”:

build_IFace_Header	build_Impl_Header
build_IFace_Debug	build_Impl_Debug
build_IFace_BasicImp	build_Impl_Imports
build_IFace_Exports	build_Impl_DLLs
build_IFace_ImplInfo	build_Impl_Constants
build_IFace_Tail	build_Impl_Impl

On the lower levels methods are still named after the matching expressions but preceded with “`write_`”.

The build functions do not work independently. They depend on their right sequential execution, because they fill certain data structures on which the next function relies.
Some of these structures directly reflect parts of the bytecodes that are referred to by bytecode references, like the name constant block. The others collect builder internal pointers to the representations of elements of the bytecode that are not reflected in one block there, but distributed over it, like for example the definition of objects.
The code for all these data structures is placed in a template called “`class IndexedCollector<`*DATATYPE*`>`”. It is used similar to an array, with the difference that elements can not be placed by the template user, but are simply passed in a call to “`.Insert(`…`)`”, where new elements are put on the top, whereas already inserted ones are detected and only their index is returned. These indices are used by the builder as values for the references within the generated bytecode. In this way, the collector helps to avoid multiple occurrences of the same data within the bytecode.

The code for the builder can be found in “ByCoBuilder.h” and “ByCoBuilder.cpp”. The reusable collectors are placed in their own files “IndexedCollector.h” and “IndexedCollector.cpp”.

2.3 Interface Loader

Like the builder, the interface loader "`class ByCoIFaceLoader`" is simply instanciated and started with a method call. When instancing, a reference to the PreStore and to an input stream carrying the bytecode interface is passed.
The method call "`loadIFace()`" reads from the stream and sets up the structures in the PreStore.
Analogously to the "`build()`" function, the work of the load function is split into parts named after the expression of the bytecode grammar, with the difference that their prefix is "`load_`". The names of the functions on lower levels are preceded with "`read_`". Examples are "`load_IFace_Exports()`" and "`read_Rule_Header()`".

The loader also uses the same "`IndexedCollector`"s, but the other way round. Those collectors that are directly representing blocks within the bytecode are filled by reading sequences of its elements from the stream. The values of references that are later read from the bytecode are used as indices into these collectors.
One "`IndexedCollector`" is special, because it has the small class "`FMLObjectPtr`" as element type, which is defined (as non-public) in the interface loader itself. It is used to hold pointers to objects that have already been loaded either from the same bytecode interface or earlier through other load processes. In the two cases, the objects are stored differently, so the type of the pointers for example is different. The class acts similar to an intelligent "`union`", hiding those internal details and differences from its user when for example only the name of the objects is needed.

The interface loader is located together with the full loader in the files "ByCoLoader.h" and "ByCoLoader.cpp".

2.4 Full Loader

The full loader is not designed as a separate class, which simply calls the interface loader, because it depends on the data structures (indexed collectors) constructed during the loading of the interface. The process of loading a full bytecode was - as described earlier - designed as an undividable unity, whereas loading the interface only is an independent sub-part.
This is reflected in the class hierarchy, where the full loader "`class ByCoLoader`" publicly inherits the interface loader and therefore is a direct extension.
It does not overload the "`loadIFace()`" method, so it could be used to load interfaces only, although this is of no direct use at the moment. The full loader is started via "`loadImpl()`", which internally uses "`loadIFace()`".

The functions are named as before, with the "load_" and "read_" prefixes.

The optional "#define BYCO_LOADER_TESTDUMP"-statement is affecting both loader classes. It activates a debug trace, which is written into the file "Loader.dbg". Currently it is only active, if "_DEBUG" is defined, too.

2.5 Integration into Flipper

As shown previously, the loader and builder are integrated into Flipper through the TParser class.
Usually, the TParser is accessed through the function "compileFile(const char *filename)", which opens the file with the given name, initialises the parser code and fills the PreStore, while it parses and checks the FML model in the file.
If it succeeds, the PreStore is committed into Flipper's stores, otherwise error messages are added to an internal error string, which is later displayed on the user interface.

Both bytecode loader and builder are started from this function.
At the beginning the type of the file is detected from the filename's extension. For full bytecodes and bytecode interfaces the appropriate loader is instanced and started. In case of an error the same error string as before is directly filled by the loader. Otherwise the structures from the PreStore are committed as usual.
If the file cannot be identified as bytecode, the parser is started and on success a newly created instance of the builder is launched, which generates the bytecode interface and implementation block from the structures the parser constructed.
The only difference the user notices in comparison to the original Flipper application is that the load dialogue for models allows to select files with bytecode extensions additionally to FML files.

To be able to generate the Flipper application as it always was or to selectively generate it with the additional bytecode loader and/or builder, two "#define" statements are used. With the definition of "_ALLOW_BYCO_LOADER" and/or "_ALLOW_BYCO_BUILDER" the matching code is activated. This concerns the loader and builder classes themselves as well as several modifications in the TParser and other parts of the Flipper code, like for example the changed file-selection mode in the load dialogue.

Because some new C++-types that have been introduced with the loader and builder, like simple structures and enumeration, are used in various parts of Flipper, they are extracted into one file "ByteCode.h". Some basic constants, like the extensions for the filenames, are kept in this file, too.

APPENDIX

The appendix consists of four main parts.
Appendix A gives additional information about the architecture of Flipper and about the Flipper Modelling Language. It is HP confidential and must therefore only be read by employees of HP and by members of the chair this thesis is written for that have a contractual relationship with the HP Laboratories in Bristol or any other relationship with HP allowing them to read HP confidential documents. It is therefore not part of the public version of this thesis, but only of the chair internal one.
Appendix B does not provide the reader with new information, but summarises the bytecode grammar described in chapter V.
Appendix C is HP confidential in the same way as appendix A and not part of the public version.
The bibliography forms the last part.

Appendix A: Details of the Flipper Application

This appendix is HP confidential and therefore not part of the official version of this thesis.

Appendix B: Summary of the Bytecode Grammar

This Appendix gives a quick summary of all BNF expressions. First the very basic and commonly used expressions are listed. Then the two bytecode blocks are presented top down.

1. Basic Expressions

On the lowest level:

Type	Byte	ShortW	Word	SmallNumber	Char
Purpose	unsigned (natural) numbers				Character
Size	8 bit	2 byte	4 byte	1 or 2 byte	1 byte

And further:

String	::=	(Char)*
Text	::=	Length String
Length	::=	SmallNumber
Number	::=	SmallNumber

The ‘SmallNumber’ needs one byte values from 0 to 127 and two bytes for the values above, up to the maximum of 32895

2. References and Types

StdRef	::=	SmallNumber

Name_Ref	::=	StdRef
RuleSig_Ref	::=	StdRef
Debug_Ref	::=	ShortW
Model_Ref	::=	StdRef
Ident_Ref	::=	StdRef
Rule_Ref	::=	StdRef
DLL_Ref	::=	StdRef
Constant_Ref	::=	StdRef
Variable_Index	::=	StdRef

‘Type_Ref’ and ‘Type_RefC’ are not going to be defined by a BNF expression, but with the aid of a table. As explained before, they are similar to the type ‘Byte’ (with restrictions on the possible values) in most of the cases and extended by an extra ‘Byte’ or ‘ShortW’ in some others. The rows with the values allowed for ‘Type_Ref’ are marked by a ‘T’ in the first column, those for ‘Type_RefC’ by a ‘C’:

Def.	Values when interpreted as ‘Byte’		Numbers of the referenced Objects	Type of following extension	Is a constant indicated?	Comment
	Binary	Hex.				
T	00000000 - 01101101	0x00 - 0x6D	0 - 109	none	no	Usual case
T	01101110	0x6E	110 - 365	Byte	no	The ext. is added to 110
T	01101111	0x6F	366 - 65901	ShortW	no	The ext. is added to 366
	01110000 - 01111110	0x70 - 0x7E	---	---	no	Reserved for ‘BuiltInType’
	01111111	0x7F	---	---	---	Reserved / Defined later
C	10000000 - 11101101	0x80 - 0xED	0 - 109	none	yes	Usual case, a constant follows in the next field
C	11101110	0xEE	110 - 365	Byte	yes	The ext. is added to 110
C	11101111	0xEF	366 - 65901	ShortW	yes	The ext. is added to 366
	11110000 - 11111110	0xF0 - 0xFE	---	---	yes	Reserved for ‘BuiltInTypeC’
	11111111	0xFF	---	---	---	Reserved / Unused

References are very often used indirectly to indicate their purpose:

Model_Name	::=	Name_Ref
Object_Name	::=	Name_Ref
DLL_Name	::=	Name_Ref

```
Function_Name        ::=  Name_Ref

Rule_Signature       ::=  RuleSig_Ref
Ident_Name           ::=  RuleSig_Ref

Const_Int            ::=  Constant_Ref
Const_Real           ::=  Constant_Ref
Const_String         ::=  Constant_Ref
Const_Object         ::=  Constant_Ref

Debug_Filename       ::=  Debug_Ref
Debug_Descr          ::=  Debug_Ref
```

The built-in types are:

```
BuiltInType          ::=  Type_Char              |  Type_Integer
                        |  Type_Real      |  Type_String
    Type_Char               ::=  0x70::Byte
    Type_Integer            ::=  0x71::Byte
    Type_Real               ::=  0x72::Byte
    Type_String             ::=  0x73::Byte

BuiltInTypeC         ::=  Type_CharC      |  Type_IntegerC
                        |  Type_RealC     |  Type_StringC
    Type_CharC              ::=  0xF0::Byte
    Type_IntegerC           ::=  0xF1::Byte
    Type_RealC              ::=  0xF2::Byte
    Type_StringC            ::=  0xF3::Byte
```

3. Debug Entries

Entries allowed in the debug area:

The first two are used in different contexts to store a filename and some other general description:

```
DI_Filename          ::=  DIFilename_ID  DIFilename_Length  DIFilename_Value
      DIFilename_ID          ::=  0x0001::ShortW
      DIFilename_Length      ::=  Length
      DIFilename_Value       ::=  String

DI_Description       ::=  DIDescr_ID  DIDescr_Length  DIDescr_Value
      DIDescr_ID             ::=  0x0002::ShortW
      DIDescr_Length         ::=  Length
      DIDescr_Value          ::=  String

DI_ThisModel         ::=  DIThisModel_ID  DIThisModel_Length
                          DIThisModel_Value
      DIThisModel_ID         ::=  0x0003::ShortW
      DIThisModel_Length     ::=  Length
      DIThisModel_Value      ::=  ( Debug_Ref )+
```

The entries within the other parts of the bytecode are:

```
DI_NoInfo            ::=  0x0081::ShortW  0x0000::ShortW

DI_Model             ::=  DI_NoInfo
                          | ( 0x0082::ShortW
                             ( 0x0002::ShortW  |  0x0004::ShortW )
                             Debug_Filename  [ Debug_Descr ]  )
      Debug_Filename         ::=  Debug_Ref
      Debug_Descr            ::=  Debug_Ref

DI_Object            ::=  DI_NoInfo  |  DI_Position  |  DI_Definition
      Debug_Position         ::=  Debug_Filename  Debug_Line  Debug_Line
      Debug_Line             ::=  ShortW

DI_Position          ::=  ( 0x0083::ShortW  0x0006::ShortW  Debug_Position )

DI_Definition        ::=  ( 0x0084::ShortW  0x0002::ShortW  Debug_Descr )

DI_Ident             ::=  DI_NoInfo  |  DI_Position  |  DI_Definition

DI_Rule              ::=  DI_NoInfo  |  DI_Position  |  DI_Definition

DI_Variable          ::=  DI_NoInfo  |  DI_Definition
```

4. Interface Block

This part defines the interface block:

```
ByCo_IFace              ::=  IFace_Header
                             IFace_TextCs  IFace_Debug
                             IFace_BasicImp  IFace_Exports
                             IFace_ImplInfo
                             IFace_Tail

IFace_Header            ::=  IFace_Magic  ByCo_Version  ThisModel_ID
      IFace_Magic             ::=  0xFBC0FACE::Word
      ByCo_Version            ::=  VersionNr  DebugType
      VersionNr               ::=  Version_Major  Version_Minor
      Version_Major           ::=  ShortW
      Version_Minor           ::=  ShortW
      DebugType               ::=  DT_NoInfo  |  DT_Concluded
      DT_NoInfo               ::=  0x00::Byte
      DT_Concluded            ::=  0x01::Byte
      ThisModel_ID            ::=  ThisModel_Name  VersionNr
            ThisModel_Name          ::=  Text

IFace_TextCs            ::=  IFace_Names  IFace_RuleSigs
      IFace_Names             ::=  Number  ( Text )*
      IFace_RuleSigs          ::=  Number  ( Text )*

IFace_Debug             ::=  Number  ( Debug_Info )*
      Debug_Info              ::=  DI_Filename  |  DI_Description  |  DI_ThisModel

IFace_BasicImp          ::=  Number  ( Model_ID  [ DI_Model ]  ObjectDefList  )*
      Model_ID                ::=  Model_Name  VersionNr
      ObjectDefList           ::=  Number  ( Object_Name  [ DI_Object ] )*

IFace_Exports           ::=  Export_Objects  Export_ObjIdents  Export_RuleHeaders
      Export_Objects          ::=  Number  ( Object_Def  [ DI_Object ] )*
            Object_Def              ::=  Object_Name  Object_Super  Object_IdentNr
                  Object_Super            ::=  Type_Ref  |  NoSuperObject
                        NoSuperObject           ::=  0x00::Byte
                  Object_IdentNr          ::=  Number
            Export_ObjIdents        ::=  Number  ( Type_Ref  Object_IdentList )*
                  Object_IdentList        ::=  ( Object_Ident  [ DI_Ident ] )*
                        Object_Ident            ::=  Ident_Name  ( Type_Ref  |
                                                     BuiltInType )
      Export_RuleHeaders      ::=  Number  ( Rule_Header  [ DI_Rule ] )*
            Rule_Header             ::=  Rule_Signature  Rule_Arity  Rule_ParTypes
                                         Rule_ParFlags  Rule_Options
                  Rule_Arity              ::=  Byte
```

```
                Rule_ParTypes          ::=  (  ( Type_Ref  |  BuiltInType )  [
                                            DI_Variable ]   )+
                Rule_ParFlags          ::=  Number  PersistentBinding
                                            ( BindFlag )*  Number  ( FuncFlag )*
                      BindFlag               ::=  Word  |  ShortW  |  Byte
                      FuncFlag               ::=  Word  |  ShortW  |  Byte
                      PersistentBinding      ::=  Number
                Rule_Options           ::=  Byte

IFace_ImplInfo         ::=  Number  ( ImplInfo_Entry )*
      ImplInfo_Entry         ::=  ImplInfo_Size  |  ImplInfo_Purpose
            ImplInfo_Size          ::=  IISize_ID  IISize_Length  IISize_Value
                IISize_ID              ::=  0x8001::ShortW
                IISize_Length          ::=  0x0004::ShortW
                IISize_Value           ::=  Word
            ImplInfo_Purpose       ::=  IIPurpose_ID  IIPurpose_Length
                                        IIPurpose_Value
                IIPurpose_ID           ::=  0x8002::ShortW
                IIPurpose_Length       ::=  Length
                IIPurpose_Value        ::=  String

IFace_Tail             ::=  Model_Creator  IFace_Hash  Impl_Hash  IFace_Signature
      Model_Creator          ::=  Text
      IFace_Hash             ::=  HashValue
            HashValue              ::=  (Byte)*16,16
      Impl_Hash              ::=  HashValue
      IFace_Signature        ::=  NoSignatureID  |  ( PGPSignatureID
                                    PGPSignature )
            NoSignatureID          ::=  0x00::Byte
            PGPSignatureID         ::=  0x01::Byte
            PGPSignature           ::=  Length  (Byte)*
```

5. Implementation Block

```
ByCo_Impl            ::=  Impl_Header
                          Impl_TextCs
                          Impl_Debug
                          Impl_Imports
                          Impl_DLLs
                          Impl_Constants
                          Impl_Impl

Impl_Header          ::=  Impl_Magic  ByCo_Version  ThisModel_ID
     Impl_Magic           ::=  0xFBC0C0DE::Word
Impl_TextCs          ::=  Number  ( Text )*
     Impl_Names           ::=  Number  ( Text )*
     Impl_RuleSigs        ::=  Number  ( Text )*

Impl_Debug           ::=  Number  ( Debug_Info )*

Impl_Imports         ::=  Import_Objects  Import_ObjIdents  Import_Rules
     Import_Objects       ::=  Number  ( Model_Ref   [ DI_Model ]
                               ObjectDefList  )*
     Import_ObjIdents     ::=  Number  ( Type_Ref  Object_IdentList )*
     Import_Rules         ::=  Number  ( Model_Ref  ImpRule_List )*
          ImpRule_List         ::=  Number  ( ImpRule_Header  [ DI_Rule ] )*
               ImpRule_Header       ::=  Rule_Signature  Rule_Arity
                                         Rule_ParTypes Rule_ParFlags

Impl_DLLs            ::=  Number  Number  ( Category_Name  DLL_List )*
     Category_Name        ::=  Text
     DLL_List             ::=  ( DLL_ID )*
          DLL_ID               ::=  DLL_Name  VersionNr

Impl_Constants       ::=  Impl_WordCs  Impl_VarLenCs
     Impl_WordCs          ::=  Number  ( Word )*
     Impl_VarLenCs        ::=  Impl_TextCs  Impl_ObjectCs
          Impl_TextCs          ::=  Number  ( Text )*
          Impl_ObjectCs        ::=  Number  ( ObjectCField )*
               ObjectCField         ::=  Number  ( Ident_Ref  Const_Any )*
                    Const_Any            ::=  Const_Char  |  Const_Int
                                              | Const_Real
                                              | Const_String  |  Const_Object
                         Const_Char           ::=  Char

Impl_Impl            ::=  Impl_CoreObjectList  Impl_MethodBodyList
                          Impl_RuleBodyList
     Impl_CoreObjectList  ::=  Number  ( Impl_CoreObject )*
```

```
          Impl_CoreObject        ::=  Type_Ref  DLL_Ref  Function_Name
                                      Function_Name  Function_Name
      Impl_MethodBodyList ::=  Number  ( Impl_MethodBody )*
          Impl_MethodBody      ::=  Rule_Ref  DLL_Ref  Function_Name
      Impl_RuleBodyList      ::=  Number  ( Impl_RuleBody )*
          Impl_RuleBody          ::=  Rule_Ref  Rule_VarDeclList  Rule_BodyExpr
              Rule_VarDeclList        ::=  Number  ( Variable_Decl  [
                                           DI_Variable ]  )*
                  Variable_Decl            ::=  ( Type_Ref  |  BuiltInType )
                                                |  (  ( Type_RefC
                                                 |  BuiltInTypeC )  Const_Any  )
              Rule_BodyExpr          ::=  Rule_IfBody  |  Rule_WhereDoBody
                  Rule_IfBody              ::=  EID_IfBody  Body_Expression
                      EID_IfBody                ::=  0xC0::Byte
                      Body_Expression         ::=  Expr_True  |  Expr_False
                                                   |  Expr_RelUse
                                                   |  Expr_Retract
                                                   |  Expr_Assert
                                                   |  Expr_Not
                                                   |  Expr_And  |  Expr_Or
                                                   |  Expr_If  |  Expr_IfElse
                                                   |  Expr_ForDo
                  Rule_WhereDoBody    ::=  EID_WhereDoBody
                                           Body_Expression
                                           Body_Expression
                      EID_WhereDoBody     ::=  0xC1::Byte
```

The expressions for the rule bodies and the expression identifications are listed separately and with less indentation, to keep them more readable:

```
      Expr_True                ::=  EID_True
      Expr_False               ::=  EID_False
      Expr_RelUse              ::=  EID_RelUse  RelationUse
      Expr_Retract             ::=  EID_Retract  RelationUse
      Expr_Assert              ::=  EID_Assert  RelationUse
      Expr_Not                 ::=  EID_Not  Body_Expression
      Expr_And                 ::=  EID_And  Number  ( Body_Expression )*
      Expr_Or                  ::=  EID_Or  Number  ( Body_Expression )*
      Expr_If                  ::=  EID_If  Body_Expression  Body_Expression
      Expr_IfElse              ::=  EID_IfElse  Body_Expression  Body_Expression
                                    Body_Expression
      Expr_ForDo               ::=  EID_ForDo  Body_Expression  Body_Expression

      EID_True                 ::=  0x01::Byte
      EID_False                ::=  0x00::Byte
      EID_RelUse               ::=  0x08::Byte
      EID_Retract              ::=  0x02::Byte
      EID_Assert               ::=  0x03::Byte
```

```
EID_Not        ::=  0x80::Byte
EID_And        ::=  0x84::Byte
EID_Or         ::=  0x85::Byte
EID_If         ::=  0x88::Byte
EID_IfElse     ::=  0x89::Byte
EID_ForDo      ::=  0x8A::Byte
```

Appendix C: Files Added to Flipper Code

This appendix is HP confidential and therefore not part of the official version of this thesis.

Bibliography

[IZUR96] The Flipper Compiler
Clem Izurieta; Investigation Report 7/96; Hewlett Packard Co.

[FERR96] The Flipper Inference Engine
Rick Ferreri; Investigation Report 1996 (Draft); Hewlett Packard Co.

[NIEL96] The Flipper Store, Caching & Database
Andrew Nielsen; Investigation Report 1996; Hewlett Packard Co.

[COMB96] Flipper modelling language
Stephan Combes; (Draft) 96; HP internal WebPage

[HEAB94] Integrated Network and Systems Management
Heinz-Gerd Hegering, Sebastian Abeck; 1994, Addison-Wesley

[CERF96] Distribution of a Management Application using Java
Stephane Cerf; Professional Thesis 1996; Hewlett Packard Co.

[MORE93] A Distributed System Management Architecture for Dolphin
Jean-Jacques Moreau; Draft #1 3/93; Hewlett Packard Co.

[BDM96] A Conceptual Approch to the Integration of Agent Technology in System Management
Domonique Benech, Thierry Desparts, Jean-Jacques Moreau; DSOM´96 10/96; Hewlett Packard Co., Institut de Recherche en Informatique de Toulouse

[GOYE96] Distributed Management by Delegation
Germán Goldszmidt, Yechiam Yemini; In Proceedings of the 15th International Conference on Distributed Computing Systems 3/96; Computer Science Department, Columbia University

[NWAN96] Software Agents: An Overview
Hyacinth S. Nwana; Knowledge Engineering Review, Vol. 11, Nr 3, Seiten 205-244, 10-11/1996; Cambridge University Press, 1996

[SCHÖ96] Distributed Network Management by Delegation - A Standarts Perspective
Jürgen Schönwälder; 10.8.1996; Department of Computer Science, University of Twente

[SCHÖ95] Distributed Network Management - Approches and Problems
Jürgen Schönwälder; Vortrag am Institit für Betriebssysteme und Rechnerverbund, TU Braunschweig, 11.4.1995; Department of Computer Science, University of Twente

[GIJA97] IBM Intelligent Agents
Don Gilbert with Peter Janca;
http://www.networking.ibm.com/iag/iagwp1.html, 1997; IBM Cooperation, Research Triangle Park, NC USA

[MULL1] Intelligent Agents for Network Operations
Nathan Muller; Datapro-Artikel 5832;

[MAEC96] Mobile Software Agents: A new Paradigm for Telecommunications Management
T. Magedanz, T. Echardt; IEEE/IFIP Network Operations and Management Symposium (NOMS), Kyoto, Japan, 15.-19.4.1996; GMD Fokus, TU Berlin

[HEFF95] Übersetzung von Programmiersprachen
Arnd Poetysch-Heffter; Skriptum zur Vorlesung - WS 1994/95 - Version 1.1; Institut für Informatik; Technische Universität München

[LLOY94] Practical Advantages of Declarative Programming
J.W. Lloyd; Invited Paper to appear on GULP-PRODE´94 Conference; Department of Computer Science, University of Bristol

[JONE95] Gofer functional programming environment Version 2.20
Mark P. Jones; An instroduction to Gofer 10/95; Department of Computer Science, Yale University

[JONE94] The implemention of the Gofer functional programming system
Mark P. Jones; Research Report YALEU/DCS/RR-1030 5/94; Department of Computer Science, Yale University

[LEPE96] Tech yourself Java in 21 days
Lemay, Perkins; ISBN 1-57521-030-4, 1996, sams net

[LIYE96] The Java Virtual Machine Specification
Tim Lindhold, Frank Yellin; ISBN 0-201-63452-X, 9/96, Addison Wesley; The Java Group

[ASLA94] Questions and Answers about MIT's Release of PGP 2.6
Hal Abelson, Jeff Schiller, Brian LaMacchia, Derek Atkins; ftp://net-dist.mit.edu/pub/PGP/PGP_FAQ, 2.6.1994; MIT

[RCF1320] The MD4 Message-Digest Algorithm
R. Rivest; Request for Comments: 1320, 4/92; MIT Laboratory for Computer Science, RSA Data Security, Inc.

[RCF1321] The MD5 Message-Digest Algorithm
R. Rivest; Request for Comments: 1321, 4/92; MIT Laboratory for Computer Science, RSA Data Security, Inc.