

GRAPHICAL PRODUCT-LINE CONFIGURATION OF NESC-BASED
SENSOR NETWORK APPLICATIONS USING FEATURE MODELS

by

MATTHIAS NIEDERHAUSEN

Diplom, University of Applied Sciences, Frankfurt, Germany, 2007

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2008

Approved by:

Major Professor
John Hatcliff

Copyright

Matthias Niederhausen

2008

Abstract

Developing a wireless sensor network application includes a variety of tasks, such as coding of the implementation, designing the architecture and assessing availability of hardware components, that provide necessary capabilities. Before compiling an application, the developer has to configure the selection of hardware components and set up required parameters. One has to choose from among a variety of configurations regarding communication parameters, such as frequency, channel, subnet identifier, transmission power, etc.. This configuration step also includes setting up parameters for the selection of hardware components, such as a specific hardware platform, which sensor boards and programmer boards to be used or the use of optional services and more. Reasoning about a proper selection of configuration parameters is often very difficult, since there are a lot of dependencies among these parameters which may rule out some other options. The developer has to know about all these constraints in order to pick a valid configuration. Unfortunately, the existing makefile approach that comes with nesC is poorly organized and does not capture important compatibility constraints.

The configuration of a particular nesC application is distributed in multiple makefiles. Therefore a developer has to look at multiple files to make sure all necessary parameter are set up correctly for compiling a specific application. Furthermore without analyzing all makefiles it is unclear what the total configurability of a nesC application is and what options and parameters are provided (e.g. is there a parameter for enabling secure communication). In addition to this, the makefile approach tends to be error-prone, since the developer has to type in variable names and values manually, that match the existing implementation. However, the existing configuration system does not capture important compatibility constraints, such as capabilities of selected hardware components.

This thesis proposes the use of feature models to configure nesC-based sensor network applications. We provide a tool-supported framework to model valid configurations and a generator that translates this model into a makefile compatible with existing nesC infrastructure. The framework automatically rules out selection of incompatible features using a build-in constraint language. Since all variables are defined in the model, misspellings of variable names are reduced and their domains are clearly defined because most variables come with all its possible options. A developer just needs to choose one or more of them by enabling certain features, where the problem of cardinality is also handled by the model. We show a detailed analysis of nesC's variability domain and how to use feature models to cover the exact behavior of nesC's makefile approach. In a following chapter we simplify our feature model and include the selection of specific hardware components, its capabilities and its dependencies. The feature model and the makefile generator offer a convenient way to configure nesC applications, that is faster, easier to understand and to handle, more transparent and once implemented it gives the possibility to adopt this configuration tool to an existing development environment.

Table of Contents

Table of Contents	v
List of Figures	vii
1 Introduction	1
2 Background Analysis	5
2.1 Example Application	5
2.2 NesC Background	7
2.3 Vision / Overview of Framework	13
2.4 Feature Modeling Tool	18
3 One-to-one feature model	20
3.1 Analyzing makerules	21
3.2 Makerules Feature Model	46
3.3 Makefile Generator	49
3.3.1 Translating single features with string attributes	51
3.3.2 Translating features representing boolean flags	52
3.3.3 Translating features with limited value choices	53
3.3.4 Building the makefile	54
3.4 Dependency Analysis	55
4 Software Product Line Configuration	64
5 Evaluation	73
5.1 Examples	73

5.2	Domain-specific feature models	78
5.2.1	Intruder Detection	80
5.2.2	Hydrology – Ground Water Monitoring	82
5.3	Assessment	84
6	Related Work	86
7	Conclusion	88
	Bibliography	92

List of Figures

2.1	Pursuer Invader Application	6
2.2	Makefiles to build an application (excerpts)	8
2.3	staged makefile structure	9
2.4	(a)generic sensor application; (b)mica mote and programmer board	9
2.5	Feature model (excerpts) for nesC node configuration	13
2.6	Overview of feature-model-based framework for sensor network configuration . . .	15
2.7	Feature Model Plugin Model Tree View	18
2.8	Feature Model Plugin Property View	19
2.9	Feature Model Plugin Constraint View	19
3.1	Feature Model of Makerules	22
3.2	Classification of Global Makerules variables	47
3.3	One-to-one feature model tree	48
3.4	Example feature model tree for XML export	49
3.5	Example of XML export produced	50
3.6	Dependencies of PFLAGS	59
3.7	Dependencies of makerules	61
3.8	Dependencies of makerules (refined)	61
3.9	Dependencies of makerules (refined) including CFLAGS	62
4.1	List of available hardware components and sensing capabilities	65
4.2	List of platform families and sensor board families	66
4.3	List of hardware component compatibilities	66
4.4	FMP tool screenshots	68

4.5	Summary of constraint categories	68
4.6	Additional feature model excerpts for nesC node configuration	69
4.7	Example feature model specialization steps	70
4.8	Feature-model-orchestrated development scenarios	71
5.1	Blink’s configuration tree and makefile output	74
5.2	MicaHWVerify’s configuration tree and makefile output	76
5.3	HighFrequencySampling’s configuration tree and makefile output	77
5.4	HighFrequencySampling’s configuration tree and makefile output	79
5.5	Architecture for Intruder Detection and Hydrology WSN product lines	80
5.6	Specialized feature model for intruder detection sensor mote	81
5.7	Specialized feature model for hydrology sensor mote	83

Chapter 1

Introduction

The past several years has seen an explosion in applications of wireless sensor networks (WSN), and the number of potential application areas seems limitless. The evolution of sensor network development infrastructure such as TinyOS/nesc and the low cost availability of hardware and network components has enabled this explosion, and has made it possible for virtually anyone to acquire the building blocks necessary to put together a sensing application.

Despite the availability and low cost of components, sensor networks can still be quite challenging to develop. At a high-level, system development must often be a *multi-disciplinary* exercise involving a variety of domain experts each having a different view of the system. For example, in a sensor network for assessing chemical content of water run-off, a hydrologist will have knowledge of water run-off patterns and specific requirements for measuring chemical make-up of water but will have little knowledge of sensor hardware or networking. An electrical engineer may be able to build water analysis sensing components, but may have little knowledge of the best way to deploy those components in the sensing field or how to assemble network resources and algorithms that ensure proper transmission of data while avoiding, *e.g.*, excessive power drains on network node batteries.

Our research group is funded by our university to act as a service organization that provides expertise and sensor network development assistance for a variety of domain areas on campus including hydrology, nuclear radiation sensing, and livestock disease control. We have found that the

ad hoc organization of existing sensor network development infrastructure and lack of high-level configuration facilities makes it difficult for different stakeholders in a sensing application to incrementally express their concerns when designing and building sensing system executables. Building executables for sensor network nodes involves choosing from among a variety of configuration options regarding hardware components (specific wireless platforms, sensor boards, programmer boards to be used), communication parameters (frequency, channels, subnet identifiers), options for particular services, etc. Reasoning about proper selection of configuration parameters and potential incompatibility between parameters can be frustrating even for experienced developers and overwhelming for experts from specific application domains who simply want to put together a sensing system with desired functionality as rapidly as possible. Challenges in proper configuration are made much more difficult by the fact that existing infrastructure for configuring builds such as the makefile framework for nesC is poorly organized, is susceptible to subtle programmer errors, fails to abstract and highlight important feature selections, and does not capture important compatibility constraints between features.

The configuration of a particular nesC application is distributed in multiple makefiles. Therefore a developer has to look at multiple files to make sure all necessary parameter are set up correctly for compiling a specific application. Furthermore without analyzing all makefiles it is unclear what the total configurability of a nesC application is and what options and parameters are provided (*e.g.*, is there a parameter for enabling secure communication).

Even if known there is a possibility to specify a certain parameter, it is still unclear how to specify this parameter to match its implementation (e.g. to specify a platform, is there a “platform” variable having a string attribute, where the string needs to match a possible value or are there multiple boolean variables/flags for each possible platform). Assuming we do know either the name of a certain feature and there is a string attribute to specify its value or the names of all boolean flags. Without an analysis we still don’t know whether we can specify multiple values separated by a certain character or enabling multiple flags at the same time or if one value excludes all others. Another problem using the makefile approach is that a developer has to type in all values and there is no check whether a value is written correctly or a typo occurred. Even the compilation

process might not recognize such an issue, e.g. an incorrect variable name leads makefiles to use the default value for the correct variable name and there is just no dependency using the wrong name. Furthermore there is no check for senseless or unreachable definitions. For example if there are multiple boolean flags for specifying one value and each one is changing the value of the same common variable. Unfortunately the related blocks of code are processed successively and so the last flag in implementation will overwrite the changes of the other flags.

Besides the problems with nesC’s makefile approach, our sensing domain partners are often interested in setting up *product lines* – families of similar sensor networks applications that perform a similar function (*e.g.*, measure properties of groundwater run-off) with commonalities in hardware selection, but variabilities in values of certain parameters (*e.g.*, the size of the sensing field, communication frequencies, specific sensor boards within families). We would like to be able to work with domain experts to design a product line, deliver a set of infrastructure components, along with a configuration facility tailored to their product family, so as to off-load to the domain experts (as much as possible) the more mundane tasks of constructing executables for a particular instance application of the product family. However, the inadequacies of the nesC infrastructure above make it difficult to place any significant configuration tasks in the hands of non-computer scientists who do not have substantial experience with nesC development.

This thesis proposes a novel framework for model-based feature selection and build configuration of nesC-based sensor network executables that addresses the above challenges. This high-level approach, based on the notion of *feature models*^{3,4,8,12,20}, supports a structured and tool-supported methodology that (a) allows multiple developers to incrementally choose characteristics of hardware and library components and (b) provides encoding and automatic checking of compatibility constraints between features to guide feature selection and to eliminate compile and execution errors associated with selection of incompatible features. The specific contributions of this work are as follows:

- we carry out an analysis of the widely-used makefile-based nesC configuration structure to identify the *choice points* associated with selecting hardware components, communication

parameters and other features needed for configuring executables for mote-based nodes,

- we develop a formal feature model that provides a taxonomy of these feature selection choice points and that encodes a variety of compatibility constraints between features,
- we illustrate how tool support for feature models provides a model-based design and configuration environment that allows developers to simultaneously specify builds following both top-down and bottom-up approaches,
- we develop a feature model compiler that translates high-level feature models down to makefiles that integrate with the existing nesC makefile infrastructure so that executables can be directly configured from feature models,
- we describe how the above feature model infrastructure provides support for developing product lines of sensor networks, and we illustrate this by overviewing two product lines constructed using the approach.

The feature model-based tool framework described above is publicly available. We believe that the type of tool support, if incorporated into the official nesC distribution, has the potential to significantly improve the distribution by providing high-level build facilities and a light-weight structured and formal approach for hardware vendors and other contributors to the distribution to expose the features and parameters offered by those contributions and the (in)compatibilities that exist with other elements of the distribution.

The research reported on in this thesis is part of a larger project on applying modeling techniques in the development of large-scale sensor networks. This thesis attempts to summarize the collective knowledge and experience of the entire research team on the specific topic of applying feature modeling techniques for sensor network configuration. I led the research and development effort on analyzing the nesC makefiles, developing a feature model for nesC makefiles, and a compiler for auto-generating makefiles from feature models as reported on in Chapter 3. Work on "high-level" feature model described in Chapter 4 was carried out jointly with other members of the research team (Todd Wallentine played a leading role in this effort). Background, Vision, and Assessment material in Chapter 2 was developed jointly with other team members.

Chapter 2

Background Analysis

This chapter provides background information about sensor networks. We need to know about the structure of nesC applications to be able to analyze its configurability. Furthermore we are describing an example application which is used in later chapters to explain certain features and the variability and commonalities of nesC applications.

2.1 Example Application

The pursuer invader application is an intruder detection and tracking application. It uses a network of sensors to detect an intruder and sends a pursuing robot to the last known position of the intruder. The network of sensors is divided into multiple sensor grids. A sensor grid consists of multiple sensors (called *motes*) forming grid with a predefined known and fixed location. Each sensor is capable of detecting an intruder by sensing light or motion activity and it has a wireless communication device attached to it which we call *base station*. Each grid is identified by a unique id (called *group id*) whereas communication of two base stations is only possible if they have the same group id. Furthermore each grid has one so called *stargate* which connects the grid to a command and control center. The sensing motes i.e. their base stations form a multihop tree rooted at the stargate. Whenever a mote detects activity it sends a message about this event to the command and control station via their corresponding stargate. If a base station cannot

communicate directly to the stargate it sends the message to another base station which is closer to the stargate and which is simply passing through such messages. The command and control center activates a pursuer robot which is moving to the last known position of the intruder. Once the pursuer robot is moving within a sensor grid it uses the pass through communication feature of this grid to communicate with the command and control center and to get information where the intruder was seen last. In order to communicate with the grid the robot has to have the same group id the grid has. Since the intruder might change its position and move to another sensor grid the pursuer is following to the new grid. To keep the communication with the command and control center alive the pursuer has to change its group id to be able to use the new grids communication features. Figure 2.1 shows the layout of the pursuer-invader application.

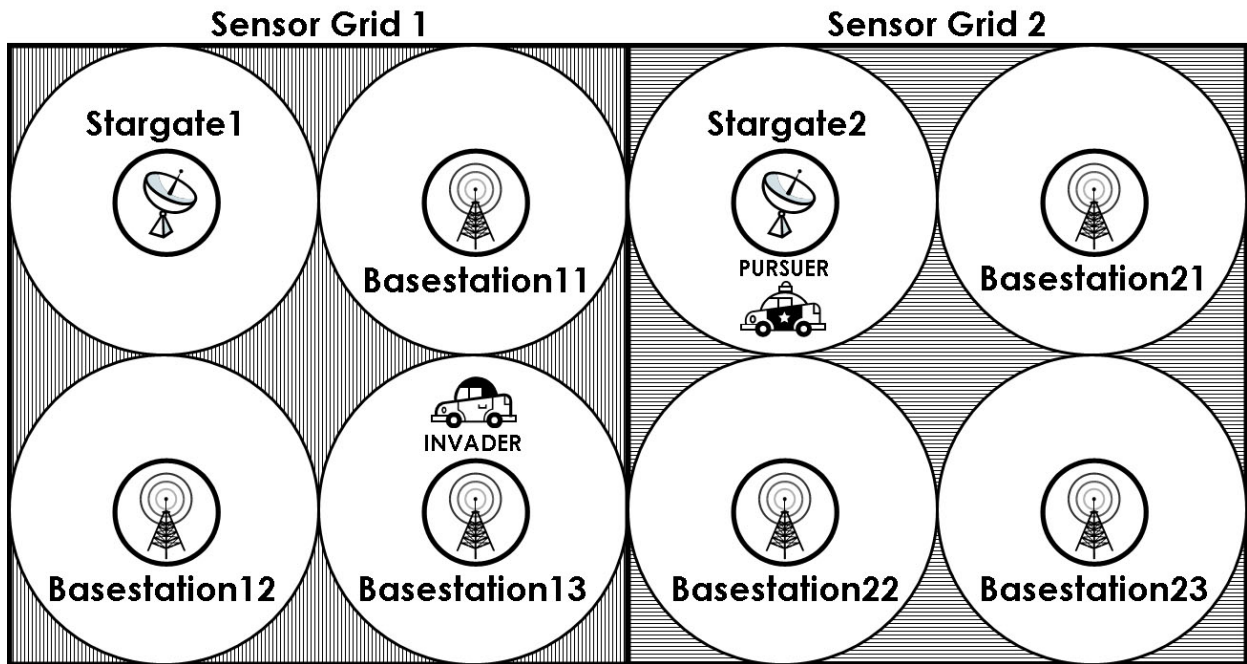


Figure 2.1: *Pursuer Invader Application*

The purpose of this application is to show the pursuer can change its wireless local group to communicate with different networks. Each network (i.e. sensor grid) has its unique default local group and each wireless device has to have the same default local group in order to be able to communicate with this sensor grid. Hence, the pursuer needs to change its local group while

moving through different sensor grids and following the invader.

To demonstrate the applications purpose it needs to consists of three different kinds of nodes:

- Stargate
- Basestation
- Pursuer

Each one of these nodes is running a different application and a different configuration. However, some configuration parameters require to be the same for all different kinds of nodes (*e.g.*, communication settings).

2.2 NesC Background

nesC¹⁰ is a programming language based on the original C programming language. It is an extension designed to support the concepts and structure of the TinyOS operating system and to develop low powered sensor network applications. The nesC compiler *ncc* uses makefiles (written in the GNU make language) to configure certain parameters before running the actual compilation, just like the original C programming language. However, a usual nesC installation includes a directory structure with sample applications which make use of multiple makefiles providing a staged concept of configurability. This staged model consists of three different makefiles:

- **global makerules**

The global makerules (or *makerules*) is a makefile that comes with the nesC developing environment. It is the top-level makefile that effects all nesC applications. It computes and provides the compiler with default options and parameters and finally executes the nesC compiler. This makefile contents a lot of variables which may or may not be used for a particular compilation and it also includes multiple dependencies among these variables. Usually an application developer shouldn't need to change the content of this makefile. Makerules is more intended to give a common construct of configurability and to define the domain of

Application Makefile	Local Makefile (MakeXbowLocal)	Global Makerules
COMPONENT=XSensorMDA300	# Settings for the default Mote Programmer,	ifndef DEFAULT_LOCAL_GROUP
SENSORBOARD=mda300	DEFAULT_PROGRAM=mib510	DEFAULT_LOCAL_GROUP := 0x7d
	MIB510=COM1	endif
XBOWROOT=%T/./contrib/xbow/tos	# default mote group	ifdef MSG_SIZE
:	DEFAULT_LOCAL_GROUP=0x88	PFLAGS := -DTOSH_DATA_LENGTH=\$(MSG_SIZE)
PFLAGS=-I\$(XBOWROOT)/interfaces		\$(PFLAGS)
-I\$(XBOWROOT)/system	# Micaz RF Power levels	endif
-I\$(XBOWROOT)/platform/\$(PLATFORM)	CFLAGS += -DCC2420_TXPOWER=0x1f	ifeq (\$(SENSORBOARD),)
-I\$(XBOWROOT)/AXStack/\$(PLATFORM)	# CFLAGS += -DCC2420_TXPOWER=0x1B	ifeq (\$(PLATFORM),mica)
-I\$(XBOWROOT)/lib	# CFLAGS += -DCC2420_TXPOWER=0x17	SENSORBOARD = micasb
:	:	endif
:	Zigbee channel selection	ifeq (\$(PLATFORM),mica2)
PROGRAMMER_EXTRA_FLAGS = -v=2	CFLAGS += -DCC2420_DEF_CHANNEL=26	SENSORBOARD = micasb
include ../MakeXbowlocal	# CFLAGS += -DCC2420_DEF_CHANNEL=25	endif
include (TOSROOT)/tools/make/Makerules	# CFLAGS += -DCC2420_DEF_CHANNEL=24	:
	:	endif
	# CFLAGS += -DCC2420_DEF_CHANNEL=11	:
	:	:

Figure 2.2: *Makefiles to build an application (excerpts)*

possible choices. It acts more like a template, whereas specific parameter values are set in one of the other makefiles, which leads to overwrite makerules default values.

- **local makefile**

The local makefile gives the second stage of nesC's makefile model. It groups together multiple applications which have something in common and use the same parameters. For example one vendor develops multiple applications which are using similar hardware products or a common protocol, then he could distribute his applications with a local makefile to make sure all required parameters are set up correctly.

- **application makefile**

The application makefile is related to one particular application. All settings in this makefile effect only one application, for example the name of an application is setup in this makefile.

Figure 2.2 shows examples of the three makefiles. The command starting the compilation process starts within the directory of one specific application. It processes the application makefile, which specifies some parameters (usually parameters which always need to be specified for any application, like name of the application) and if this application belongs to a group with common settings (e.g. frequency setting for communication like Crossbow or vendor specific settings like XMesh) it will include the local makefile. After that it will give up control to the global makerules file, which comes with TinyOS and does most of the dependency computation and default settings

for all applications. Finally makerules will start the compiler with a complete list of parameters. In the process settings of makerules (default values) are overwritten by local makefile ones, which are overwritten by application ones. In addition to the makefiles one can specify parameters explicitly at the command line, which are also processed by makerules. Figure 2.3 shows the staged organization of the nesC makefile structure and the order of processing.

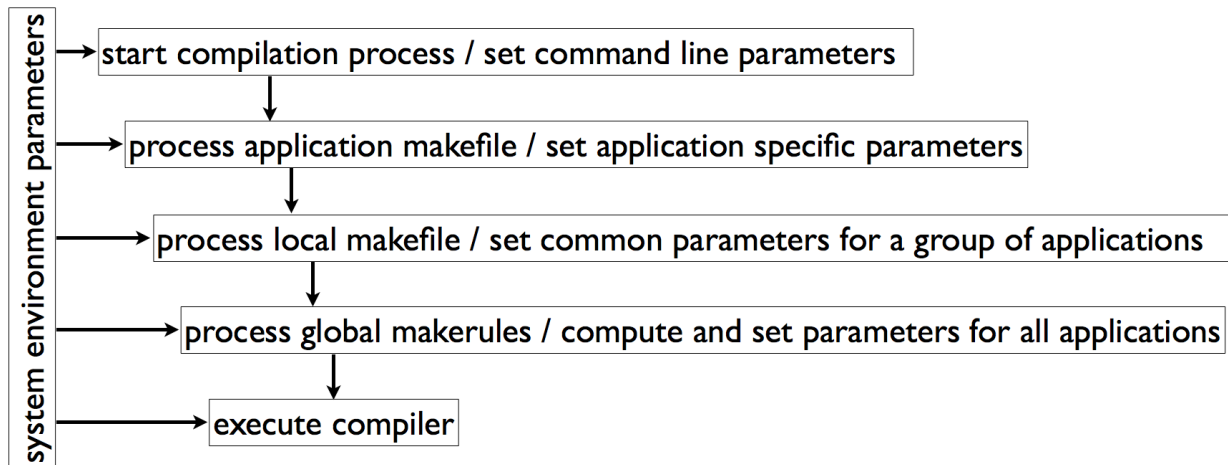


Figure 2.3: *staged makefile structure*

Sensor applications are often data-centric in nature where the main task is to collect sensed data, process it at intermediate nodes, and deliver it to consumers for possible actions. Usually a sensor network application consists of multiple nodes (motes) forming a network by communicating to each other.

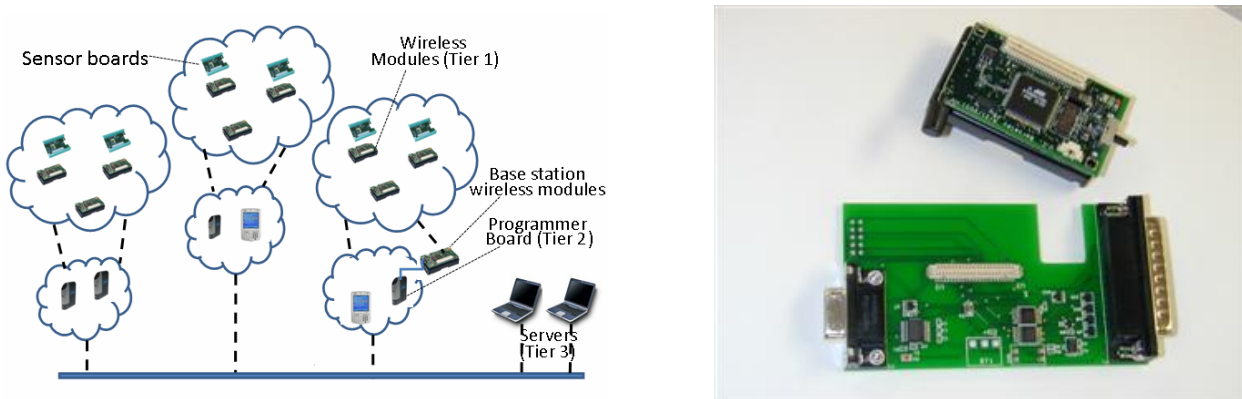


Figure 2.4: (a) *generic sensor application*; (b) *mica mote and programmer board*

Figure 2.4(a) shows the architecture of a typical sensor application which consists of a number of components at different tiers¹¹. Tier 1 entities include wireless modules (platforms), some of which may have in-built sensors or attached sensor boards. The task of the wireless modules is to relay sensor readings to Tier 2 entities. The sensor field must be instrumented with sensors to collect data for the desired attributes which are communicated via the wireless modules. Depending on the scale and nature of the sensor field, the system may have to be decomposed into sub-systems (or groups). Tier 2 consists of gateways and handheld devices with a richer set of computation and communication capabilities. Each such node is associated with a separate sub-system, and has a wireless module attached to communicate with Tier 1 modules in the sub-system. Tier 2 nodes may perform intermediate fusion and analysis on the data before forwarding it to Tier 3 entities (PCs and servers).

Figure 2.4(b) shows a mote at the top and its programmer board at the bottom from the mica platform family. The programmer board can be connected to a PC using either serial or parallel interface. To upload a compiled application image to a mote, the mote can be attached to a programmer board, that supports the motes hardware platform. Some programmer boards support multiple platforms like mica, mica2 or mica128 by having different connectors.

The existing mechanisms to design and build the executables for a sensor applications has a number of limitations which hinder their rapid development. In particular, novice designers and domain engineers (who are not sensor network experts) typically have to devote significant efforts to develop new applications. The shortcomings include the following:

- *Lack of infrastructure to make available choices explicit:*

A large number of choices exists for selecting wireless modules, sensor boards, gateways, and software libraries. There is a lack of infrastructure to organize the available choices and make them available in a systematic manner to a designer. At present, a designer has to rely on past experience and product datasheets to identify the possible choices.

- *Lack of mechanisms to ensure consistency among choices:*

The design of a sensor application may involve a team of experts. As the design moves

through the different stages, a number of decisions may have to be made at different stages by different team members. This can result in the following consistency issues:

- *Consistency across variability points:*

Given multiple variability points, a choice made at one point may limit the choices available at other places. For example, a choice of a sensor board may limit the available choices for platform boards and vice-versa. Currently, there is a lack of tools that can help a team of designers in avoiding conflicting choices.

- *Consistency across various stages of development:*

The design of a sensor application may go through several phases, and choice options exercised in one phase may have to be consistent with those in subsequent phases or may have to be reflected in a parameter setting in a later phase. For example, if a particular sensor board, say MDA300, is selected during the hardware selection phase, then the team member coding the system must ensure that the variable `Sensorboard` is set to MDA300 in the makefiles. Currently, there is a lack of tools to formally capture and document such design decisions, and to reflect them in other stages. No mechanism is currently available to identify an incorrect configuration of the makefiles other than compilation errors or incorrect functioning of the application.

- *Consistency across makefiles:*

The configuration information for an application is distributed across multiple makefiles. Therefore, the developer has to look at multiple files to ensure that all parameters have been set up correctly. Furthermore, a user of an already designed and configured application has to analyze all of the makefiles to determine how the application was configured, and and to identify the options and parameters left open for the user to configure. Even when a configurable option is known, it is often unclear how to exercise this option (e.g. to specify a platform, it is unclear whether there is a single “platform” variable to be assigned a platform name or multiple boolean variables for each possible platform).

– *Manual generation and configuration of makefiles:*

Presently, the developer is responsible for manually typing in the parameter values in the makefiles, and there is no check to ensure consistency. For example, even typographical errors may result in a successful compilation (an incorrect variable name specified in an application makefile may lead to the usage of the default value in the global makefile). Furthermore, there is no check for inconsistent or unreachable definitions. For example, there could be multiple boolean flags for specifying the same option. The blocks of code containing these boolean flags are processed successively, and the last flag to be processed will overwrite the others.

– *Consistency across nodes and tiers:*

In setting up a network, one may have to deploy several Tier 1 wireless modules. For wireless modules to communicate, the frequency range used for communication by these modules must match. Thus, one must ensure that matching wireless platforms are selected in setting up a Tier 1 network. For example, a Mica2 board cannot communicate with a MicaZ board. However, in order to configure large scale networks, the notion of group-ids has been incorporated. This allows a network to be sub-grouped into sub-nets, where each sub-net is assigned a different group-id. Therefore, we only need to ensure that modules belong to be same group are able to communicate with one another. The designer also has to deal with compatibility issues across tiers. For example, the selection of a wireless platform may limit the choice of sensor boards and the gateways and vice-versa. For example, the Stargate Netbridge gateway currently (at the time of writing of this thesis) does not support TelosB as a base-station module attached to it, but can potentially communicate with TelosB boards in Tier 1 via a MicaZ board.

2.3 Vision / Overview of Framework

We seek to overcome the limitations and problems of nesC’s existing makefile configuration mechanism by using feature models. Feature models allow us to model the complete variability domain of an application graphically and to limit the model to important points of choice, where a developer has to make a decision. In addition, feature models usually support constraint mechanisms, to rule out incompatible feature selections automatically.

A number of different graphical feature model notations have been developed, which focus on different aspects of feature selections^{3,4,8,12,20}. The *Orthogonal Variability Model* (OVM) developed by Pohl et al.²⁰ is one approach how to model variability information. Its notation introduces ‘variability points’ representing points of choice and ‘variants’, that represent possible values and uses a graphical model based on standard UML diagrams. These UML diagrams show dependencies between variability points and variants and may also specify relationships between different UML diagrams.

One other approach is the *Cardinality-based Feature Modeling* (CBFM) notation of Czarnecki et al.⁸, which supports stepwise specialization of feature models.

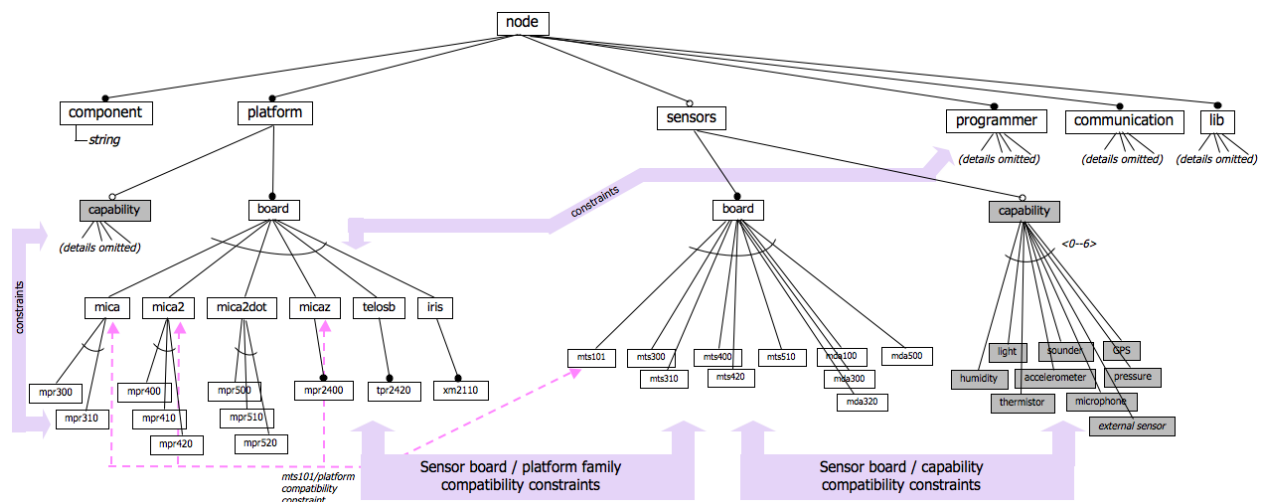


Figure 2.5: Feature model (excerpts) for nesC node configuration

Figure 2.5 illustrates some of the basic elements of the CBFM notation using fragments of a much larger feature model that we use for configuring nesC node executables. Czarnecki and

Eisenecker define a feature “as a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family”, and thus “a feature may denote any functional or non-functional characteristic at the requirements, architectural, component, platform, or any other level.”⁷. While modeling based on this broad definition of “feature” can be useful in WSN development, for our work on better supporting WSN executable configuration, we focus on features that represent choice points exposed to developers in the current nesC build process for a single node.

A feature diagram is a tree of features with the root representing some concept or system element. In our case, the root represents a single WSN node (e.g., mote). Each node in the feature diagram may contain *sub-features*. For example, ‘node’ has a number of sub-features including ‘component’, ‘platform’, ‘sensors’ etc., while ‘sensors’ has ‘board’ and ‘capability’ sub-features. The hierarchical arrangement of subfeatures also indicates that the selection of a specific feature (e.g., ‘sensors’) causes additional feature choices (e.g., ‘board’ and ‘capability’ to be exposed). The features listed so far are all *solitary features* – single features that are either selected or not. Cardinality constraints for solitary features are represented by circles: \circ (e.g., on ‘sensors’) represents the cardinality $\langle 0 - 1 \rangle$ which specifies that the feature is optional¹; \bullet (e.g., on ‘platform’) represents the cardinality $\langle 1 - 1 \rangle$ which specifies that the feature is mandatory. *Feature groups* capture situations in which choices must be made from a group of features. For example, it is mandatory to select a platform board which must be drawn from a feature group providing available families of boards ‘mica’, ‘mica2’, etc. Feature groups have an associated group cardinality $\langle n - m \rangle$ specifying the minimum n and maximum m features that can be selected from the members of the group covered by an accompanying arc. If an explicit group cardinality is not given, $\langle 1 - 1 \rangle$ is assumed. For example, if the optional ‘sensors’ feature is selected, exactly one sensor board from among the group ‘mts101’, ..., ‘mda500’ must be selected. The $\langle 0 - 6 \rangle$ group multiplicity² on the optional ‘sensors capability’ captures the fact that our model allows developers to optionally specify capabilities of sensors boards, and model constraints cause the choices for sensor boards

¹Sensorless motes are often used for communication purposes.

²The upper bound of 6 on the multiplicity is somewhat arbitrary; it’s based on the fact that we know of no sensorboard supported by nesC that has more than 6 on-board sensors.

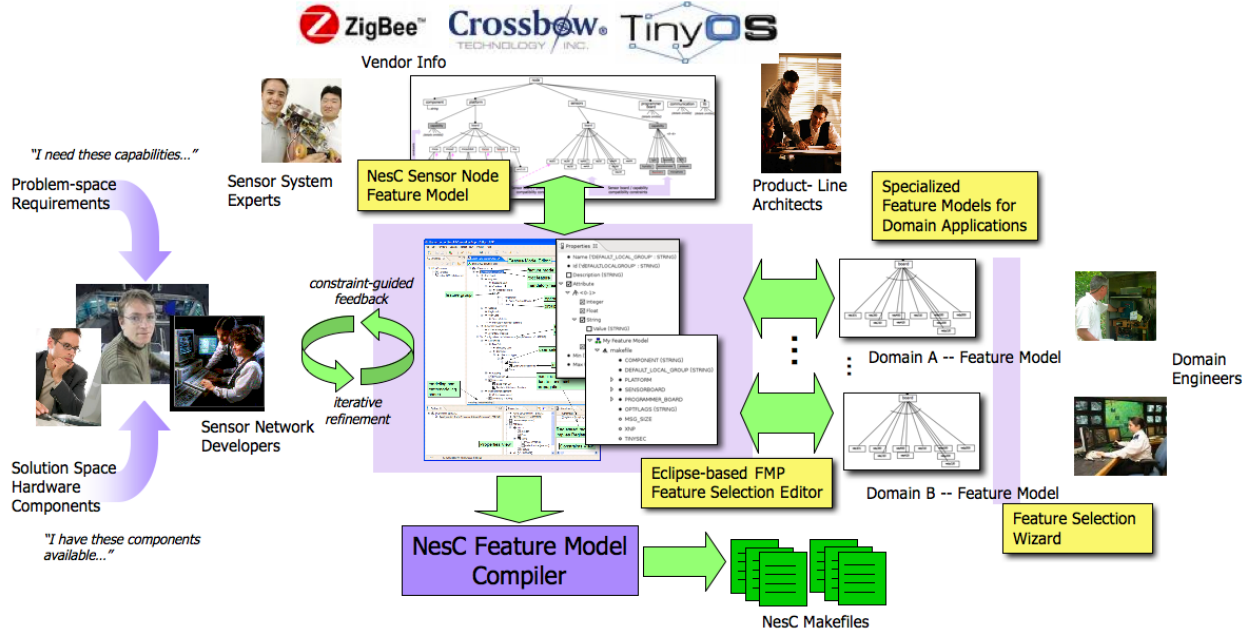


Figure 2.6: Overview of feature-model-based framework for sensor network configuration

presented to the developer to be narrowed automatically to only the boards that support the selected sensor capabilities.

The capture of constraints between features is an important aspect of realistic modeling of product lines and complex features sets. We are using the Eclipse-based FMP from Czarnecki’s group at the University of Waterloo¹⁹, and it allows constraints to be specified in the XML XPath language. Our full feature model contains over 60 constraints; the model excerpt in Figure 2.5 highlights the existence of constraint sets between sensor boards and platform families (indicating that a sensor board is only compatible with certain families), and between sensor board capabilities and sensor boards (indicating that the specific sensor types supported by each board). As a specific example, Figure 2.5 shows a constraint stating that the ‘mts101’ sensor board is only compatible with ‘mica’, ‘mica2’, and ‘micaz’ platforms.

It is important to note that we are not proposing the feature model presented in this thesis as *the canonical model* to use for configuring mote executables; other feature model organizations may be just as effective. We simply aim to illustrate the substantial benefits of the feature model approach to configuration via a feature model organization that we have found useful.

Figure 2.6 presents an overview of the tool capabilities that we have developed and illustrates some of the primary workflows associated with the tools. The foundation of framework is a formal feature model that captures at a high level of abstraction (a) the choices and configuration options currently exposed in nesC makefiles, (b) features capturing specific sensing and networking capabilities not accounted for in the makefiles, and (c) constraints between features (also not accounted for in makesfiles). The feature model is originally designed and maintained by sensor network development experts and by product-line architects who arrange the presentation of features to facilitate the specification of builds for families of sensor network applications. The mechanisms by which vendors expose particular libraries and accompanying options in nesC makefiles are very ad hoc, and vendor-provided documentation regarding hardware compatibilities is often omitted or is presented in a confusing way. While we have constructed our feature model ourselves after a careful examination of nesC makefiles and documentation from a variety of vendors, an open source feature-model-based configuration mechanism packaged with the nesC distribution could provide an excellent means for vendors to systematically expose configuration options and to formally document hardware compatibility constraints in a framework that provides helpful visualizations of such features/constraints and that directly drives the build process.

The FMP tool supports feature model definition, editing, and feature selection based on a particular feature model. The left side of Figure 2.6 describes how feature selection enhances a conventional nesC development workflow. Instead of working with cumbersome makefiles that contain many distracting details, FMP guides teams of developers in an iterative selection of features organized by concepts. As particular hardware components are selected, FMP’s management of formally specified compatibility constraints causes remaining feature options that would lead to incompatible hardware combinations to be disabled. This automatically avoids many classes of configuration errors. The bidirectionality of constraints allows developers to proceed with feature selection in an order which best suits their needs: selecting a particular platform family first disables incompatible sensor board choices, selecting a sensor board first disables incompatible platform families. Feature selection can be staged and refined as development proceeds. For example, a platform family (*e.g.*, ‘mica2dot’) can be selected (which, via constraints, narrows other

choices), but the selection of a particular platform (*e.g.*, ‘mpr510’) can be postponed. Instead of only reasoning about features in the solution space (*e.g.*, available hardware components), configuration can also be guided by selecting from among *capabilities* (*e.g.*, light and temperature sensing) needed in the problem space. Again, the constraint model disables selection of features (*e.g.*, sensor boards) that do not provide the needed capabilities. This type of functionality is quite valuable for letting multiple team members and domain experts simultaneously express feature selections related to hardware and desired capabilities – the constraint mechanism guarantees consistency across variability points and development stages. It is possible for tool support to highlight mandatory choice points where selections have not yet been made, and to notify the developer when selections are complete and ready to direct a build.³ We have integrated a feature model compiler into FMP that translates completed instances of our sensor network feature model into nesC makefiles – enabling developers to execute a build from within the high-level feature model framework.

The right side of Figure 2.6 describes how feature modeling enables the construction of build scenarios for families of similar sensor networks and supports domain engineers who are not sensor network experts in configuring their systems. As sensor network product-line architects receive requirements from engineers working in one of any number of sensing domains, they make initial selections of capabilities, perhaps choose hardware families, and develop application code. Application code is delivered to domain engineers along with partially instantiated feature models that only expose choices yet to be made and documents selections that were already made in the product-line development stage. When domain engineers are ready to deploy their systems, a feature selection wizard processes a partially instantiated feature model and guides them through the selection of remaining mandatory features and creates (via the feature model compiler) and executable that can be downloaded to motes. This dramatically reduces the level of knowledge about nesC development that domain engineers must possess to deploy a functioning sensor network.

³FMP does not currently provide this functionality, but it would be straightforward to add.

2.4 Feature Modeling Tool

In this thesis we use the notation of feature models, because there is a tool publicly available¹⁹ to model and parse these feature models.

Feature models use a proprietary tree structure to organize ‘features’ and ‘feature groups’ as its nodes, where features represent certain aspects of a software application, that may or may not be a variability point. Feature groups specify cardinality constraints among a group of features, e.g. a car has exactly one color but may be equipped with a four-wheel drive or not.

The tool is a plugin for the eclipse 3.2 development environment and uses three panels to display all information. The main window (figure 2.7) shows the feature model diagram and allows to edit basic properties of the model, like changing feature names or the organization of feature nodes.

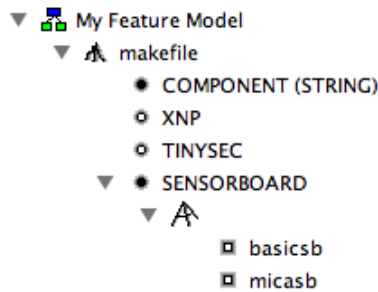


Figure 2.7: *Feature Model Plugin Model Tree View*

The feature model tree uses the following symbols for different kinds of nodes:









-  root feature node
-  mandatory feature node
-  optional feature node
-  feature group, that allows only one selected child node
-  feature group, that allows one or more selected child nodes
-  feature group with user-specified cardinality constraints
-  optional child node of a feature group
-  mandatory child node of a feature group

Figure 2.8 shows the property window of a feature or a feature group, where advanced properties like cardinality constraints of feature groups or attributes of features can be changed.

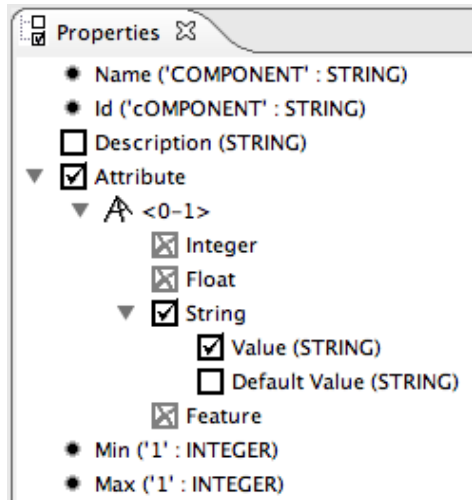


Figure 2.8: *Feature Model Plugin Property View*

Specifying additional constraints, which are not cardinality restrictions, is possible using the constraint window, shown in figure 2.9. The constraint mechanism for feature models uses a subset and the notation of the XML Path Language (XPath)⁵, defined by the World Wide Web Consortium (W3C).

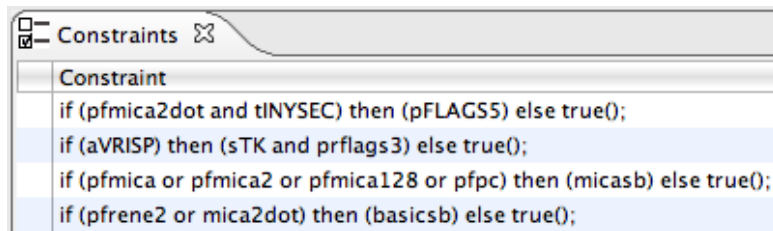


Figure 2.9: *Feature Model Plugin Constraint View*

These constraints allow to automatically activate or deactivate feature nodes from a feature model configuration. The feature model contains a model, that forms a pattern of a feature tree, and configurations, that show an instance of a subtree of the model. Only within a configuration features can be activated or deactivated, but constraints are defined for the whole model. A constraint affects all instances of the model (i.e. all configurations).

Chapter 3

One-to-one feature model

Designing a feature model and support for product lines usually begins with a *variability analysis*. This analysis identifies *commonalities*, that all systems to be developed have in common and *variabilities*, choice points or aspects for which systems to be developed can be parameterized. The natural starting for a variability analysis in our work is the collection of makefiles that come with the nesC distribution. These makefiles represent the current means by which developers express choices in building executables. In this section, we examine the makefile variables, determine which of these represent actual choice points for configuration or reflect product line aspects, and look for dependences that may exist between variables. We then assess the extent to which makefile variables effectively expose choice points and constraints between choice points to developers. Based on this assessment, we propose an initial feature model that enables feature selection at the same level of abstraction as variables in the existing makefiles.

There are different ways to start modeling the top level of the makerule file. Simplified, it is one file with many variables. Starting with one feature which is representing the whole makerules file, you can add variables (features) directly under this top level or you can put a feature group between these two levels. Since we don't care about how many features are selected in a configuration of this model, there is no need for a feature group. So we start developing a feature model tree with a root and one feature for each variable occurring in makerules. We will get an exact one-to-one mapped feature model to the original makerules.

3.1 Analyzing makerules

We start developing a one-to-one feature model by analyzing all variables of makerules. Temporary variables which are defined and used only within makerules without any exterior dependencies do not represent point of choice. Therefore, we do not model them in the feature model. We need to find each variable which might get its value from outside of makerules, to analyze its purpose, its domain and its dependencies. Finally, we can develop a way to cover each of those variables and its dependencies using a feature model.

MAKERULES

This variable gives the location (pathname) of the global makerules file.

```
ifndef MAKERULES
    MAKERULES=$(shell ncc -print-tosdir)/../apps/Makerules
endif
```

This statement is a simple definition of a default value. Whenever a user doesn't care about this variable it will be declared and it will have a default value. The default value is calculated by a shell command and gives a pathname to the default global makerules file, which should be the currently processed file. In the feature modeling framework we are using, it is possible to model default string values for a feature. In the modeling part of the feature modeling framework, you can define a default value, and if a user is not defining a different value of this feature in the configuration area, the default value would be set. Where there is a "ifndef" block in the makerules file with no other commands in it other than those that define a variable, this can be modeled as a single mandatory feature with a default string value in the modeling area. It has to be mandatory to make sure that there is always a variable with this name - no matter which value.

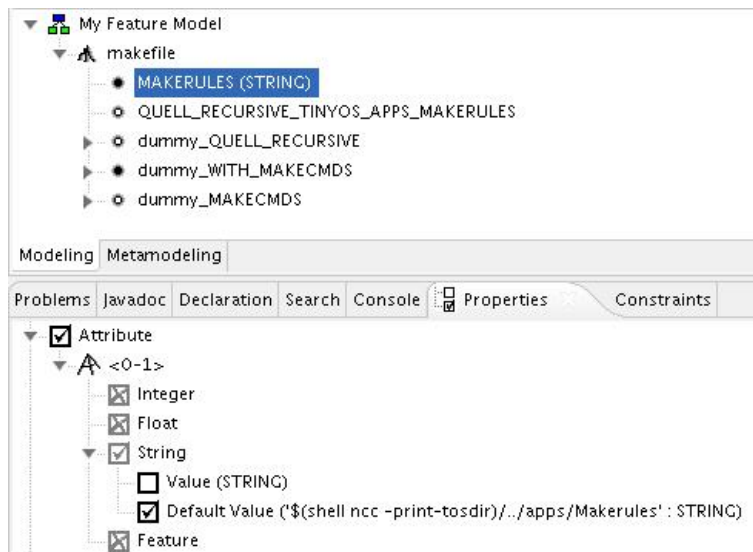


Figure 3.1: *Feature Model of Makerules*

QUELL_RECURSIVE_TINYOS_APPS_MAKERULES

This is a control variable to stop recursive inclusion of makesrules file.

```
ifndef QUELL_RECURSIVE_TINYOS_APPS_MAKERULES
    QUELL_RECURSIVE_TINYOS_APPS_MAKERULES=1
    include $(MAKERULES)
else
    ... remaining of file
```

This is not just a definition of a variable because there is a second line and an else block that controls execution of the whole build process. This variable works together with the formerly defined “MAKERULES” variable and they decide which makerule file should be processed by the compiler. Spelling to a true/false table there are four possible control flows, where two of them are identical:

	MAKERULES defined	MAKERULES undefined
QUELL_R... defined	process else block of current file	process else block of current file
QUELL_R... undefined	process user defined makerules file (stop processing current file)	process default makerules file (stop processing current file) (usually the default value should point to the currently opened makerules file and the compiler will start over processing the current file, but now with defined “MAKERULES” and “QUELL_R...” variables and finally it will jump into the else block.)

Of course in case where the user defines another makerules file it is not possible to model this unknown process. The three other cases most likely end in the else block of the current makerules file. The variable itself is modeled as a simple optional feature without any attributes but unfortunately there is no way to model the exact behavior of makerules in this case. The idea to have an optional feature (here “dummy_QUELL_RECURSIVE”) which includes a feature group simulating the else block comes close to the original behavior. This optional feature depends on QUELL_RECURSIVE_TINYOS_APPS_MAKERULES - if this variable is defined the dummy variable is set and if it is not defined the dummy block is not set and all definitions inside this block/feature-group wouldn’t be considered. For getting such a behavior an additional constraint has to be defined as follows:

```
if (qUELLRECURSIVETINYOSAPPSMAKERULES)  
then (dummyQUELLRECURSIVE)  
else true();
```

The name starts with ‘dummy_’ because the makefile generator implementation usually treats all feature groups as a variable with multiple value entries added together to a single string with spaces (explained later when used). The generator needs the ability to decide the purpose of a certain feature group, and therefore it is coded in the generator that all features starting with ‘dummy_’ are ignored but their sub-items are processed as usual. The else block of QUELL_RECURSIVE_TINYOS_APPS_MAKERULES includes the remaining makerules files. Therefore all of the following variables are within this else block and modeled as sub-features of dummy_QUELL_RECURSIVE.

DEFAULT_LOCAL_GROUP

Since nesC is designed for sensor network applications, we can assume each mote needs to communicate with other motes via a wired or wireless network. This variable is used to divide those networks into sections (e.g. sensor grids in our example application), where communication is only possible within the same section. Motes that need to communicate with each other have to have the same group id. If the developer doesn’t specify a group id, this part of makerules will assign a default value for it. In this case it produces one network for all motes, thus, communication is possible among all motes by default.

```
ifndef DEFAULT_LOCAL_GROUP  
    DEFAULT_LOCAL_GROUP := 0x7d  
endif
```

This part of makerules is just a default value definition like in “MAKERULES”. It can be modeled the same way as a mandatory feature with a string attribute and a predefined default value in the modeling area.

OPTFLAGS

The OPTFLAGS variable also starts with a default value definition. Unlike the ones we have seen before, its value can be changed depending on other values. The value of OPTFLAGS is read only once at the end of makerules as a parameter for the final compilation step.

```
ifndef OPTFLAGS
    OPTFLAGS := -Os
endif
```

This part could be modeled the same way like the default value definitions before, as a mandatory feature with a default string value. But there are other variables which might influence the value of OPTFLAGS.

```
ifeq ($(PLATFORM), pc)
    OPTFLAGS := -g -O0
:
ifeq ($(DBG)_x, debug_x)
    OPTFLAGS := -O1 -g -fnesc-no-inline
:
ifeq ($(DBGOPT)_x, debugopt_x)
    OPTFLAGS := $(OPTFLAGS) -g
:
:
```

The first if statement in this block checks whether the variable PLATFORM has the string value “pc”. The other two statements are appending “_x” at the value of DBG and DBGOPT and checking if this created value is the same like “debug_x” or “debugopt_x” strings. When OPTFLAGS is read there are eight possible sources of its value:

- completely user defined value
- default value
- the value defined in the PLATFORM block
- the value defined in the DBG block

and each one of these four can be appended with the string “-g” if DBGOPT is set. Since the values in the PLATFORM and DBG blocks already include a “-g” in their definition there is no need to append it a second time. To model this structure in a feature model we need a feature OPTFLAGS. Instead of specifying an attribute with default value for this feature we add a feature group and in this group we add one feature for each possible source with a predefined string value. Therefore we start with four features plus the DBGOPT as a fifth feature with “-g” as its value. With cardinality of the feature group set to “1..2” we can only activate 2 features at a time. Putting together the string has to be postponed to the generator step that parses the model. But we need to add several constraints to make sure user value, default value, DBG value and PLATFORM value are exclusive. Because user value and default value are overwritten by PLATFORM value or DBG value it is sufficient to add those constraints. Furthermore we need constraints enforcing dependencies like “if the PLATFORM is set to “pc” the correct OPTFLAGS value has to be set”.

```

if (pPLATFORMpc) then (oPTFLAGSpatform) else true();
if (dDBG) then (oPTFLAGScdbg) else true();
if (dDBGOPT) then (oPTFLAGScdbgopt) else true();
if (oPTFLAGSpatform) then ((not oPTFLAGScuser) and
(not oPTFLAGScdefault)) else true();
if (oPTFLAGScdbg) then ((not oPTFLAGScuser) and (not oPTFLAGScdefault)
and (not oPTFLAGSpatform)) else true();

```

If the DBG variable is set it also overwrites previous definitions from the PLATFORMS block, because the DBG block is implemented after the PLATFORMS block in makerules.

NESC_FLAGS

Makerules defines NESC_FLAGS as variable a default string value. Its content whether user defined or default value is read only once for a definition of PFLAGS which always holds.

```

ifndef NESC_FLAGS
    NESC_FLAGS := -Wnesc-all
endif

```

This variable is modeled as a mandatory feature with a string attribute and a predefined default value in the modeling area. Since it is always used to form a part of PFLAGS value, we add NESC_FLAGS under the feature group of PFLAGS.

BASEDIR

The definition of BASEDIR is also a default value assignment. Instead of assigning a string, it executes a shell command which determines a certain directory and stores the pathname in BASEDIR. Its value is only read once for defining the value of DOCDIR.

```

ifeq ($(BASEDIR)_x, _x)
    BASEDIR := $(shell pwd | sed 's @\(.*\)/apps.*$$@1@' )
endif

```

Since the result of the executed shell command might be different at the model time than at compilation time, we cannot execute the shell command from within the feature model. Instead we can model this variable the same way as other default value definitions. Once make processes its value, it will run the shell command.

DOCDIR

The DOCDIR variable represents the directory where generated documentation files should be stored in. NesC provides an application called “nesdoc” which generates documentation files similar to Javadoc. This is the only usage of DOCDIR in makerules.

```

ifeq ($(DOCDIR)_x, _x)
    DOCDIR := $(BASEDIR)/doc/nesdoc
endif

```

This is another way to check if a variable is already defined by user (i.e. other makefiles) and to express a default value definition. The definition of DOCDIR uses the former specified variable BASEDIR.

PLATFORMS, PLATEAUX

The PLATFORMS variable is part of a staged configuration. It has a default value specifying a domain of possible platform families currently supported by the nesC distribution. Its default value is a list of four possible platform families separated with a space character.

```
ifeq ($(PLATFORMS)_x, _x)
    PLATFORMS = mica mica2 mica2dot pc
endif
```

This variable could be modeled as a feature group the four defined platform families as its child features. Since PLATFORMS specifies the domain of possible values and there is a later step choosing one out of those for which the application is going to be compiled. Modeling PLATFORMS is not necessary if we model the more specific choice.

Once PLATFORMS is defined its value is copied to the variable PLATEAUX and the string “all” is appended in PLATEAUX. This makes it possible for the developer to choose ‘all’ instead of one particular platform, which runs the compiler for each possible platform.

OBJCOPY, SET_ID, PROGRAMER, NCC and LIBS

These five variables are simple value definitions, that name string constants used in multiple places in the makefile. These definitions allow the value of each string constant to be defined in one place.

```

OBJCOPY      = avr-objcopy
SET_ID       = set-mote-id

PROGRAMMER   = uisp

NCC          = ncc

LIBS         = -lm

```

There is no need to include these variables in the feature model because they are always set.

MSG_SIZE

This variable sets the maximum data length of each message. If it is defined, it sets a parameter including itself as part of PFLAGS.

```

ifdef MSG_SIZE

    PFLAGS := -DTOSH_DATA_LENGTH=$(MSG_SIZE) $(PFLAGS)

endif

```

This states a constraint which adds a string to the value of “PFLAGS” whenever a user or a former included makefile (local makefile, application makefile) already defined a “MSG_SIZE” variable - no matter which value it has. MSG_SIZE is modeled as an optional feature without any attributes. In addition to the feature there is a constraint defined:

if (mSGSIZE) then (pFLAGSmsg) else true();

Since it is not possible to access the value of MSG_SIZE from within the feature model constraint and copy its value to PFLAGS the copy process is postponed to execution of make.

XNP and XNP_DIR

The XNP variable is a boolean flag which determines whether network reprogramming capability should be activated or not. XNP_DIR is a fixed string variable pointing to the directory containing network reprogramming related files.

```

XNP_DIR := ../../tos/lib/Xnp
ifdef XNP
    PFLAGS := -I$(XNP_DIR)
    ... $(shell $(XNP_DIR)/ident.pl .ident_install_id $(COMPONENT))
    ... $(PFLAGS)
endif

```

If XNP is defined, it will add two strings to the value of PFLAGS. XNP_DIR doesn't need to be modeled and XNP is modeled as an optional feature with no attributes in the modeling area which can be either enabled or disabled. An additional constraint is defined which adds the two strings to PFLAGS value:

if (xNP) then (pFLAGsxnp) else true();

TINYSEC, TINYSEC_KEY, KEYFILE and KEYNAME

The TINYSEC variable is a boolean flag determining whether secure communication features should be enabled or disabled. KEYFILE and KEYNAME are fixed string variables used for computing an encryption key.

```

ifndef TINYSEC
    TINYSEC := false # default:  disable tinysec
endif

```

This first part of TINYSEC is just a default value definition. Even if this variable is supposed to be a boolean flag it is modeled as a feature with a string attribute. Since there are no further steps handling the false case of this variable and since there is no attribute for boolean values the predefined default value is “true” and it is modeled as an optional feature. If this value is not set by user (‘false’ case) it will be ignored because the feature is not enabled and if it is set its value would be ‘true’ and a constraint is taking care of this case.

```

ifeq ($(TINYSEC),true)
    TINYSEC_KEY := $(shell mote-key -kf $(KEYFILE) -kn $(KEYNAME))
    :
    PFLAGS := $(PFLAGS)
    ... -I%T/lib/TinySec
    ... -I%T/platform/%p/TinySec
    ... -DTINYSEC_KEY="$(TINYSEC_KEY)"
    ... -DTINYSEC_KEYSIZE=8
    ifeq ($(PLATFORM),mica2dot)
        PFLAGS := $(PFLAGS) -I%T/platform/mica2/TinySec
    endif
endif
endif

```

Once TINYSEC is set it runs a shell command to generate a key used for secure communication. Afterwards it modifies PFLAGS by adding TINYSEC related strings. Since TINYSEC_KEY is not used anywhere else and since it has a fixed value we don't need to model it. For the case TINYSEC is enabled we need to have a constraints making changes to PFLAGS:

```

if (tTINYSEC) then (pPFLAGStinysec) else true();
if (pfmica2dot and tTINYSEC) then (pPFLAGStinysecmica2dot) else true();

```

MIB510, MIB510_ and MIB5100

MIB510 is one specific programmer board used for loading compiled application images to a mote. The variable MIB510 is a string attribute determining whether this programmer board is currently used or not and at which serial port to find the programmer board. MIB510_ and MIB5100 are temporary helper variables to extract MIB510 out of all command line parameters.

```

ifneq (x$(MIB510),x)
    PROGRAMMER := STK
    PROGRAMMER_FLAGS=-dprog=mib510
    ... -dserial=$(MIB510)
    ... $(PROGRAMMER_PART)
    ... $(PROGRAMMER_EXTRA_FLAGS_MIB)
endif

```

There is a constraint for modifying values of PROGRAMMER and PROGRAMMER_FLAGS. Since it is not possible to define values from within a constraint, PROGRAMMER and PROGRAMMER_FLAGS have to be modeled as feature groups and their possible values are separate sub-item features of this group with predefined values. A constraint can modify values of PROGRAMMER and PROGRAMMER_FLAGS by selecting or deselecting the sub-items. MIB510, EPRB and AVRISP are defining values of the same variables PROGRAMMER and PROGRAMMER_FLAGS. Because of that it is assumable that these three variables are excluding each other. The only possibility to exclude two of these features is by modeling PROGRAMMER_FLAGS as a feature group with three sub-items (for values depending on MIB510, EPRB and AVRISP) and to set the maximum of selected features to ‘1’.

<i>if (mIB510) then (sTK and prflagsmib) else true();</i>

EPRB, EPRB_ and EPRB0

EPRB is an programmer board for loading application images to a mote using ethernet. The variable EPRB is a string attribute determining whether this programmer board is currently used or not and at trough which host-id (i.e. ip address) it is reachable. EPRB_ and EPRB0 are temporary helper variables to extract EPRB out of all command line parameters.


```

ifneq (x$(EPRB),x)
    PROGRAMMER := STK
    PROGRAMMER_FLAGS=-dprog=stk500
    ... -dhost=$(EPRB)
    ... $(PROGRAMMER_PART)
    ... $(PROGRAMMER_EXTRA_FLAGS_STK)
endif

```

There is a constraint for modifying values of PROGRAMMER and PROGRAMMER_FLAGS. Since it is not possible to define values from within a constraint, PROGRAMMER and PROGRAMMER_FLAGS have to be modeled as feature groups and their possible values are separate sub-item features of this group with predefined values. A constraint can modify values of PROGRAMMER and PROGRAMMER_FLAGS by selecting or deselecting the sub-items.

MIB510, EPRB and AVRISP are all defining values of the same variables PROGRAMMER and PROGRAMMER_FLAGS. Because of that it is assumable that these three variables are excluding each other. The only possibility to exclude two of these features is by modeling PROGRAMMER_FLAGS as a feature group with three sub-items (for values depending on MIB510, EPRB and AVRISP) and to set the maximum of selected features to ‘1’.

if (ePRB) then (sTK and prflagsavriscp) else true();

AVRISP

AVRISP is a variable that defines whether the In-System Programming (ISP) feature of the AVR system should be used to upload compiled application images to a mote⁶. The variable AVRISP is a string attribute determining whether this programmer board is currently used or not and at which serial port to find the programmer. In contradiction to MIB510 and EPRB the value of AVRISP is not extracted from the command line parameters.

```

ifdef AVRISP
    PROGRAMMER := STK
    PROGRAMMER_FLAGS=-dprog=stk500
    ... -dserial=$(AVRISP)
    ... $(PROGRAMMER_PART)
    ... $(PROGRAMMER_EXTRA_FLAGS_AVRISP)
endif

```

There is a constraint for modifying values of PROGRAMMER and PROGRAMMER_FLAGS. Since it is not possible to define values from within a constraint, PROGRAMMER and PROGRAMMER_FLAGS have to be modeled as feature groups and their possible values are separate sub-item features of this group with predefined values. A constraint can modify values of PROGRAMMER and PROGRAMMER_FLAGS by selecting or deselecting the sub-items.

MIB510, EPRB and AVRISP are all defining values of the same variables PROGRAMMER and PROGRAMMER_FLAGS. Because of that it is assumable that these three variables are excluding each other. The only possibility to exclude two of these features is by modeling PROGRAMMER_FLAGS as a feature group with three sub-items (for values depending on MIB510, EPRB and AVRISP) and to set the maximum of selected features to ‘1’.

<i>if (aVRISP) then (sTK and prflagsavrisp) else true();</i>
--

AVRISP_DEV

If the In-System Programming feature of the AVR system should be used, i.e. the variable AVRISP is defined, the compiler needs to know how to access the AVR programmer. AVRISP_DEV specifies the serial port that has to be used to communicate with the AVR programmer. The value of AVRISP_DEV is not defined or modified in makerules, but used to define the value of PROGRAMMER_FLAGS_INP.

```
ifdef AVRISP
    PROGRAMMER_FLAGS_INP=-dprog=stk500 -dserial=$(AVRISP_DEV) ...
endif
```

PROGRAMMER_EXTRA_FLAGS_MIB,

PROGRAMMER_EXTRA_FLAGS_STK and

PROGRAMMER_EXTRA_FLAGS_AVRISP

These variables are used whenever their corresponding programmer board was chosen. None of them has any definitions in makerules and they are only used once to copy their content to the PROGRAMMER_FLAGS variable. They are modeled as separate features without any attributes.

SENSORBOARD

SENSORBOARD represents families of sensor and data acquisition boards which can be attached to certain platforms. If not specified, makerules will assign a default sensorboard according to the platform chosen with the following statements:

```

ifeq ($(SENSORBOARD),)
    ifeq ($(PLATFORM),mica)
        SENSORBOARD = micasb
    endif
    ifeq ($(PLATFORM),mica2)
        SENSORBOARD = micasb
    endif
    ifeq ($(PLATFORM),mica128)
        SENSORBOARD = micasb
    endif
    ifeq ($(PLATFORM),rene2)
        SENSORBOARD = basicsb
    endif
    ifeq ($(PLATFORM),pc)
        SENSORBOARD = micasb
    endif
    ifeq ($(PLATFORM),mica2dot)
        SENSORBOARD = basicsb
    endif
endif

```

If the platform chosen is *mica*, *mica2*, *mica128* or *pc* makerules will take the *micasb* sensor board and if the platform is *mica2dot* or *rene2* it will take the *basicsb* sensor board to be the value of `SENSORBOARD`. We model this variable as a feature group having two children “*micasb*” and “*basicsb*”, where the cardinality allows only one to be enabled at a time. To model the default assignment we need additional constraints.

```

if (pfmica or pfmica2 or pfmica128 or pfpc) then (micasb) else true();
if (pfrene2 or mica2dot) then (basicsb) else true();

```

PFLAGS

PFLAGS is the most central and most important variable in makerules. The value of PFLAGS is modified several times while processing makerules. Its value is a string consisting of multiple strings separated by space characters. Some of those strings are always included in PFLAGS, some of them are optional and some are included by constraints. There is no initial definition of PFLAGS, all strings are added to the former value of PFLAGS, therefore it might carry values which are defined outside of makerules. The intention of PFLAGS is to collect all choices, combine them in one string and pass this string to the compiler at the end of makerules.

Some values or strings are always part of PFLAGS, starting with its initial value and adding the following strings in some part of makerules:

- -Wall
- -Wshadow
- -DDEF_TOS_AM_GROUP=\$(DEFAULT_LOCAL_GROUP)
- \$(NESC_FLAGS)
- -target=\$(PLATFORM)
- -board=\$(SENSORBOARD)
- -fnesc-cfile=\$(BUILDDIR)/app.c

Other values or strings are only included in PFLAGS if a certain variable exists:

- If MSG_SIZE exists include:
 - -DTOSH_DATA_LENGTH=\$(MSG_SIZE)
- If XNP exists include:
 - -I\$(XNP_DIR)
 - \$(shell \$(XNP_DIR)/ident.pl .ident_install_id \$(COMPONENT))

A third group of values or strings are included in PFLAGS if a certain variable carries a certain value:

- If the value of TINYSEC is ‘true’, include:
 - -I%T/lib/TinySec
 - -I%T/platform/%p/TinySec
 - -DTINYSEC_KEY="\$(TINYSEC_KEY)”
 - -DTINYSEC_KEYSIZE=8
- If the value of TINYSEC is ‘true’ and the value of PLATFORM is ‘mica2dot’, include:
 - -I%T/platform/mica2/TinySec
- If the value of PLATFORM is ‘pc’, include:
 - -pthread -fnesc-nido-tosnodes=1000
- If the value of PLATFORM is anything else than ‘pc’, include:
 - -finline-limit=100000

By modeling PFLAGS as a feature group, that can have multiple sub-features activated at a time, we can introduce one separate sub-feature for each value. Activating such a feature includes its constant string into PFLAGS value. Thus, we can state constraints on these sub-features to cover all above dependencies and influence PFLAGS value from the constraint language. But we also need to access all the values from the value dependent conditions, to determine whether a condition is satisfied. Therefore PLATFORM and TINYSEC have to be modeled as feature groups, too, with at least as many sub-features to differentiate between the given conditions, e.g. PLATFORM has to have at least two sub-features representing ‘mica2dot’ and ‘pc’.

```

if (mSGSIZE) then (pFLAGS1) else true();
if (xNP) then (pFLAGS2) else true();
if (pfmica2dot and tINYSEC) then (pFLAGS5) else true();
if (tINYSEC) then (pFLAGS10 and tINYSECKEY) else true();
if (pfpc) then (pFLAGS7 and (not pFLAGS9)) else true();
if (pfmica or pfmica2 or pfmica128 or pfrene2 or pfmica2dot or pfall) then (pFLAGS9
and (not pFLAGS7)) else true();

```

CFLAGS

There is nothing that depends on CFLAGS and nothing in makerules that might influence CFLAGS value or usage. In fact this variable occurs only once in makerules. It is used as one parameter when executing the final compilation step. According to the reference manual of the make language, this variable is commonly used by developers to pass parameters directly to the compiler.

COMPONENT

This variable is a unique string, that identifies each application and gives its name. It is also used to name the output file of the compiler. Depending whether XNP is set, COMPONENT is also part of PFLAGS, but this is the only dependency on COMPONENT.

HOME

The HOME variable helps to access the users directory. It is a system environment variable that points to the users home directory, no matter which operating system is currently used. Since it is a system variable makerules doesn't modify its value.

```
KEYFILE := $(HOME)/.tinyos_keyfile
```

This variable is only be used once for defining value of KEYFILE. KEYFILE belongs to TINYSEC feature, that enables secure communication. Makerules used HOME to get the path that contains a file called “.tinyos_keyfile”.

PROGRAMMER_PART

The value of PROGRAMMER_PART is included in PROGRAMMER_FLAGS in all cases, that is used for installing the application on a mote, using a certain programmer board. The value of PROGRAMMER_PART depends on the value of PLATFORM. There are

four sections in makerules for the four platforms ‘mica’, ‘mica128’, ‘mica2’ and ‘mica2dot’:

```
ifeq ($(PLATFORM), mica)
    PROGRAMMER_PART=-dpart=ATmega103 --wr_fuse_e=fd
endif
ifeq ($(PLATFORM), mica128)
    PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
endif
ifeq ($(PLATFORM), mica2)
    PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
endif ifeq ($(PLATFORM), mica2dot)
    PROGRAMMER_PART=-dpart=ATmega128 --wr_fuse_e=ff
endif
```

The fact that the use of PROGRAMMER_PART occurs earlier than its definition in makerules, might be a little bit confusing. First it is used to define PROGRAMMER_FLAGS and later on its value is computed depending on PLATFORM. After its definition the value is never used again. However, the language make provides two different ways of defining variables, which are called *flavors*⁹. In the way PROGRAMMER_FLAGS is defined, it postpones reading the value of PROGRAMMER_PARTS till PROGRAMMER_FLAGS is used. This happens only in the final compilation (i.e. installation) step. Therefore this unusual order of using variables and assigning values to them still affect the installation step.

PROGRAMMER_EXTRA_FLAGS

PROGRAMMER_EXTRA_FLAGS may provide some additional parameters, which will be included in PROGRAMMER_FLAGS. If AVRISP is not defined, the value of PROGRAMMER_EXTRA_FLAGS will also be included in PROGRAMMER_FLAGS.INP. This definition is divided into the same four platform sections as in PROGRAMMER_PART.

BUILDLESS_DEPS, BUILD_EXTRA_DEPS, LDFLAGS

These three variables are not defined in makerules nor modified. Furthermore there are no other variables that depend on the existence of these variables or on their values. All three are directly passed to the final compiler step.

PROGRAMMER_FLAGS_INP

The value of `PROGRAMMER_FLAGS_INP` depends on the platform and whether `AVRISP` is defined. It is computed in a way similar to the computation of `PROGRAMMER_PART` and has four sections for the platforms ‘mica’, ‘mica128’, ‘mica2’ and ‘mica2dot’. Each section has a condition whether `AVRISP` is defined or not. If `AVRISP` is not defined `PROGRAMMER_FLAGS_INP` includes the value of `PROGRAMMER_EXTRA_FLAGS` plus a string that indicates a parallel programming device.

```
PROGRAMMER_FLAGS_INP=-dprog=dapa $(PROGRAMMER_EXTRA_FLAGS)
```

If `AVRISP` is defined makerules assign a different value that doesn’t include `PROGRAMMER_EXTRA_FLAGS`.

```
ifdef AVRISP    PROGRAMMER_FLAGS_INP=-dprog=stk500 -dserial=$(AVRISP_DEV)
               -dpart=ATmega128 endif
```

In this case the serial in-system programming device of the AVR system is used. The last part “-dpart=ATmega128” is used for the platforms ‘mica128’, ‘mica2’ and ‘mica2dot’. For the ‘mica’ platform it uses “ATmega103” instead.

PLATFORM

This variable is not defined in makerules, it is a parameter extracted from the command line when attempt to run the compiler. There are many dependencies on this variable. The extraction is made by using value of `PLATFORMS` variable as a pattern for filtering out its value from the command line. Possible values for this variable are: ‘mica’, ‘mica2’, ‘mica128’, ‘renee2’, ‘pc’, ‘mica2dot’ and ‘all’. Although this variable is treated as a string with

whitespace separators it seems that it has to have one and only one value because all the dependencies in makerules are comparing its value if it matches (and not includes) one string.

USAGE

In case there is a mistake detected while processing makerules, make spits out an error message that show how to use it properly. This error message is defined in makerules as the value of variable USAGE.

```
define USAGE
Usage:  make <platform>

make all

make clean

make install[.n] <platform>

make reinstall[.n] <platform> # no rebuild of target

make docs <platform>

Valid platforms are:  $(PLATFORMS)
endef
```

MAKECMDGOALS, MAKE

Both of these variables are part of the make language. MAKE represents the application make and can be used to execute another make command from within the currently processed one. MAKECMDGOALS provides all command line parameters which were given by the developer when the current make process started. It is treated as a string variable with space character to separate parameters. Makerules uses this variable to keep track which parameters were already processed. Each time makerules processes a certain section that is related to a command line parameter, it removes this parameter/string from MAKECMDGOALS. Before starting the compiler this string should be empty, otherwise makerules wasn't able to

use this parameter and outputs the error message stored in USAGE.

PROGRAMMER_FLAGS

The value of PROGRAMMER_FLAGS depends on the operating system and on the programming device used, i.e. which of the variables ‘MIB510’, ‘EPRB’ or ‘AVRISP’ is defined. First of all its default value is assigned that includes the values of PROGRAMMER_PART and PROGRAMMER_EXTRA_FLAGS and assumes a parallel programming device. For the FreeBSD operating system one more string is added to this value.

```
ifeq ($(shell uname),FreeBSD)
    PROGRAMMER_FLAGS=-dlpt=/dev/ppi0 -dprog=dapa $(PROGRAMMER_PART)
    $(PROGRAMMER_EXTRA_FLAGS)
else
    PROGRAMMER_FLAGS=-dprog=dapa $(PROGRAMMER_PART)
    $(PROGRAMMER_EXTRA_FLAGS)
endif
```

Then makerules checks if one of the three programming boards ‘MIB510’, ‘EPRB’ or ‘AVRISP’ is used and overwrites the default value of PROGRAMMER_FLAGS. This new value still includes PROGRAMMER_PART, but instead of the PROGRAMMER_EXTRA_FLAGS the new value includes one of the programmer board specific variables like PROGRAMMER_EXTRA_FLAGS_MIB. Furthermore the ‘dprog’ string is changed to a serial device “mib510” or “stk500” instead of the parallel “dapa”.

```

ifneq (x$(MIB510),x)
    PROGRAMMER_FLAGS=-dprog=mib510 -dserial=$(MIB510) $(PROGRAMMER_PART)
    $(PROGRAMMER_EXTRA_FLAGS_MIB)
endif
ifneq (x$(EPRB),x)
    PROGRAMMER_FLAGS=-dprog=stk500 -dhost=$(EPRB) $(PROGRAMMER_PART)
    $(PROGRAMMER_EXTRA_FLAGS_STK)
endif
ifdef AVRISP
    PROGRAMMER_FLAGS=-dprog=stk500 -dserial=$(AVRISP) $(PROGRAMMER_PART)
    $(PROGRAMMER_EXTRA_FLAGS_AVRISP)
endif

```

PROGRAMMER

PROGRAMMER determines whether a parallel or serial programmer board is used. The default value is “DAPA” that indicates a parallel device. If one of the programmer board variables MIB510, EPRB or AVRISP is set, it will overwrite the default with “STK” indicating a serial device.

```

PROGRAMMER := DAPA
ifneq (x$(MIB510),x)
    PROGRAMMER := STK
:

```

BOOTLOADER

The two platforms ‘mica2’ and ‘mica2dot’ require a specific boot loader. If one of these platforms is used the variable BOOTLOADER will be defined with a string pointing to the

corresponding boot loader image file.

BUILDDIR, MAIN_EXE, MAIN_SREC

These three variables are statically defined path- and filenames.

```
BUILDDIR = build/$(PLATFORM)
MAIN_EXE = $(BUILDDIR)/main.exe
MAIN_SREC = $(BUILDDIR)/main.srec
```

MAIN_TARGET

MAIN_TARGET is either a copy of MAIN_EXE or MAIN_SREC depend on which platform is used. If this compilation should produce a PC application it chooses the .exe file otherwise it takes the .srec file.

DBG, DBGOPT and DOCS

These three variables do not influence the compilation process. They are all extracted from the command line parameters and all of them are boolean flags. If DBG and DBGOPT are set the compilation outputs debug information and if DOCS is defined an application called ‘nesdoc’ is executed that generates a documentation, similar to javadoc for java files.

3.2 Makerules Feature Model

After analyzing all variables occurring in makerules we provide a feature model that covers the domain of variability of makerules in this section. This feature model forms a tree structure where all variability points of makerules can be configured graphically. Since all variables and their values are predefined in this tree, misspellings or the problem of undefined parameters can be reduced. Using this model, we provide a generator that is capable of writing one makefile that combines all the parameters of the three nesC makefiles in a later section. With a few changes this generator may also be used to start the compilation process directly from within the development environment.

The root of the feature model is a single node called “makerules”. This node has one child feature for each variable occurring in makerules that represents a variability point. Since not all of the 57 makerules variables can be influenced by the developer, some of them will not show up in the feature model.

The three variables HOME, MAKECMDGOALS and MAKE are special variables representing features of the make language itself. Even if MAKECMDGOALS is influenced directly by the command line parameters, a developer doesn’t specifically define this variable. Seven other variables (OBJCOPY, SET_ID, PROGRAMER, XNP_DIR, NCC, LIBS and KEYNAME) are string constants. These variables exist, because they are helpful in writing and using makerules. They are defined in one place and used several times in makerules. It would be easy to model these variables as mandatory features with a predefined default string value, but their value does not change, so they do not represent a variability point. To keep the feature model simple we do not include them in the model and postpone their definition to the generated output makefile. Figure 3.2 gives an overview of the different types of variables (explained in chapter 3.4).

Model 3.3 shows the complete feature model tree of makerules variables. The features are grouped into three parts, indicated by names that start with “dummy_”. The reason for having such dummy features is the upcoming makefile generator. This generator has to differentiate between features that represent makefile variables and others that don’t. When the generator

#	Purpose/Role of Variables
3	hold information for MAKE instructure (<i>e.g.</i> , home directory, name of specific make application used)
7	string constants (<i>e.g.</i> , name of nesC compiler, library directory)
4	temporary variables for extracting values from the command line
1	string displayed for MAKE errors that states how to invoke MAKE
2	used for recursive inclusion of makerules
9	variables pointing to files or directories calculated from other variables
1	result of an executed shell command
4	not modified nor used but passed to the final compiler step (<i>e.g.</i> , CFLAGS mechanism for allowing application dependent flags)
3	turn on/off debugging and documentation generation
2	define set of possible platforms
2	variables completely computed from others by the MAKE to consolidate arguments to pass to compiler
#	Purpose/Role of Variables intended to be set by developers
4	default value definitions <i>e.g.</i> , DEFAULT_LOCAL_GROUP
2	boolean flags (XNP, TINYSEC)
5	hardware component selection
5	extra flags for programmer boards
1	specification of message size (MSG_SIZE)
1	specification of main component (COMPONENT)
1	variable which may have a user-defined initial value but while processing makerules strings are added to it (PFLAGS)

Figure 3.2: *Classification of Global Makerules variables*

parses the feature model it will ignore all features starting with “dummy_” and proceed with either child or sibling nodes. The first group “dummy_MAKECMDS” includes parameters that are originally extracted from the command line. The second group “dummy_WITH_MAKECMDS” includes variables that depend on or that are necessary for handling command line parameters. The third group “dummy_QUELL_RECURSIVE” includes all variables that are not related to command line parameters and their basic computation.

In addition to this feature model tree, there are several constraints defined, that enforce the original behavior of makerules. Unfortunately, the constraint language implemented in this feature modeling tool is not powerful enough to state constraints accessing feature attributes values. It is not possible to read a feature’s string attribute or to assign a new string value from within the constraint language. As a workaround, we make use of feature groups instead. Most variables that change their values have a string value that represents multiple string values separated by space characters. Those variables are modeled as a feature with a feature group as their only child. The child nodes of this feature group represent string values that might be part of the variables complete string value. These child nodes do not have a string attribute, but they are named with the associated string. We can then easily state constraints or enable and disable those value

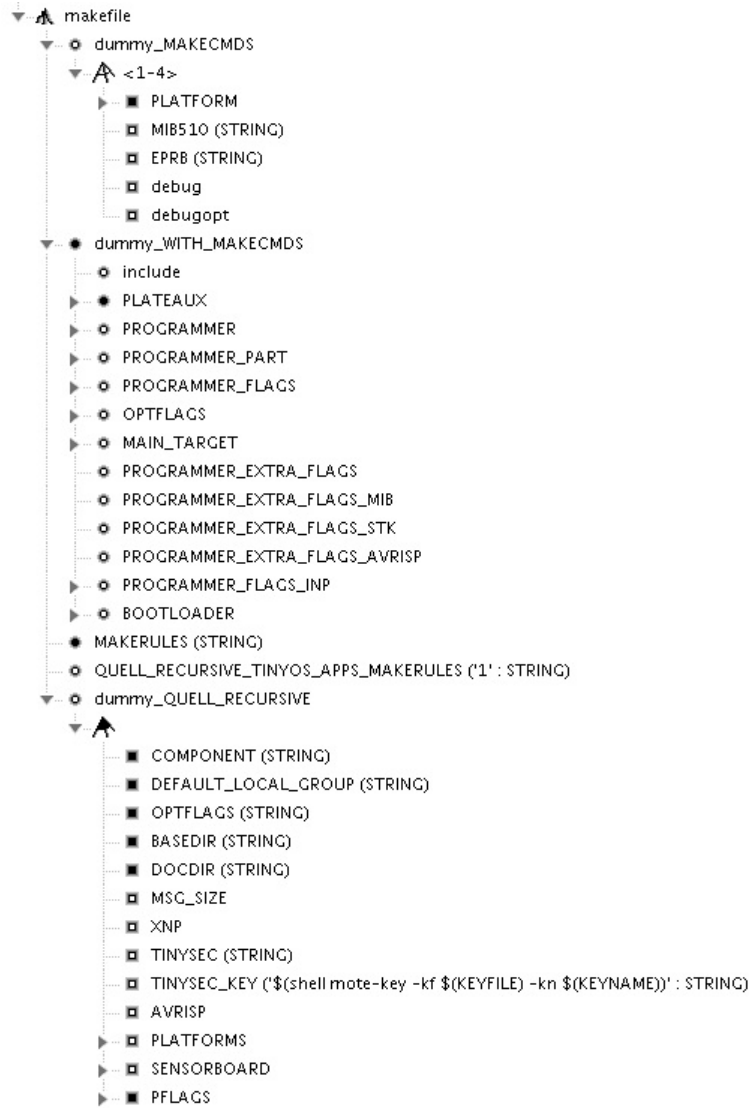


Figure 3.3: *One-to-one feature model tree*

features from within the constraint language. The generator can find feature groups and appends all activated value features to one string and assigns this string to the variable represented by the group's parent node.

3.3 Makefile Generator

The makefile generator parses a feature model and produces a makefile based on makerules with all different kinds of parameters. This one makefile replaces all three makefiles nesC usually uses, but it produces the exact same configuration. Hence, a compilation using only this one makefile produces exactly the same application image that the standard nesC mechanism would produce. One advantage of the feature model is it first checks all feature group cardinality constraints and all additional coded constraints before it produces an output. Using this behavior we can enforce constraints like hardware compatibility constraints that were not checked by nesC's approach by defining additional feature model constraints.

Developing the generator started with the in-build feature model tree XML-Export feature implemented in the tool we are using. This export feature parses the feature model tree and translates the tree into a XML string including names and values of activated features. Figure 3.4 shows an example feature model tree and figure 3.5 shows the generated string by applying the XML export function to the configuration node.

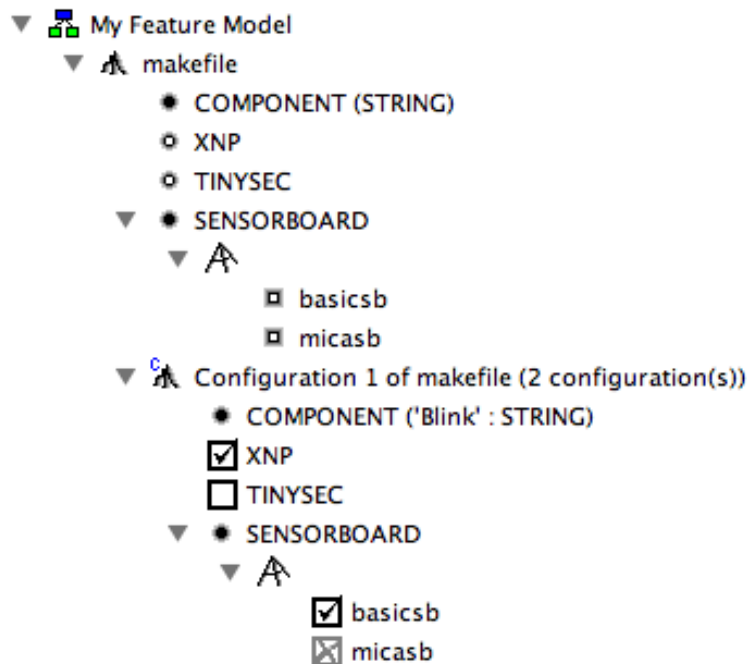


Figure 3.4: Example feature model tree for XML export

```
<feature name="makefile" type="NONE" id="makefile">
  <feature name="COMPONENT" type="STRING" value="Blink" id="cOMPONENT">
  </feature>
  <feature name="XNP" type="NONE" id="xNP">
  </feature>
  <feature name="SENSORBOARD" type="NONE" id="sENSORBOARD">
    <feature name="basicsb" type="NONE" id="basicsb">
    </feature>
  </feature>
</feature>
```

Figure 3.5: *Example of XML export produced*

The important information we need to generate a makefile are “name”, “value” and “type”. We use the “name” value to represent the name of a variable in the new makefile and the “value” string to represent the value of this variable. As we can see in the generated XML export the XNP feature is included in the export string but it does not have a value. Unfortunately the feature model tool does not support boolean data types for activating and deactivating features. All these features are of type “NONE”, except the COMPONENT feature that does have an attribute of type “STRING”. We need to find another way of determining whether a feature represents a boolean variable like XNP or choice of predefined single or multiple strings.

Since the source code of the feature model tool is publicly available can duplicate the XML export feature as a new context menu item and change its implementation. The context menu is defined in the class `FmpActionBarContributor.java` in the package `ca.uwaterloo.gp.fmp.presentation`. There we copy the call of the XML export function as a new “generate Makefile” function to the context menu of all node instances. Furthermore we copy the class `XMLExport.java` and rename it to `GenerateMakefile.java` in the package `ca.uwaterloo.gp.fmp.provider.action`. This new class includes the parsing algorithm of feature model trees and behaves like the XML export function initially. We then need to modify its implementation to generate makefiles instead generating XML strings.

The generator takes the feature from where the command was applied to and parses this feature and its complete subtree recursively. A node can either be an instance of the feature or the feature

group class, but the initial node from where the generator starts has to be a feature. Feature nodes within a configuration have a config state property that allows to identify whether this feature is activated or not in the tree model. There are five possible states a feature node can have:

- ☒ USER_SELECTED
- ☒ USER_ELIMINATED
- ☒ MACHINE_SELECTED
- ☒ MACHINE_ELIMINATED
- ☐ UNDECIDED

In the beginning of a feature model configuration all feature nodes are Furthermore, each feature has a ‘min’ and ‘max’ value, where the ‘max’ value of a single feature node is always set to ‘1’. If a features ‘min’ value is ‘0’, the feature is considered optional, and if the ‘min’ value is ‘1’ the feature is mandatory. The generator processes a feature node and its subtree only if:

- it is a mandatory feature
- is has the config state USER_SELECTED or
- is has the config state MACHINE_SELECTED

As we mentioned earlier, we want to have a features that we can use for organizing the model tree, but that are not translated into makefile variables. This is done by excluding all features whose name starts with “dummy_”, so the generator checks this property and proceeds processing the current node if the name starts with anything else. There is one more exception needed for not translating the root node, that has the name “makefile”.

3.3.1 Translating single features with string attributes

Simple variable definitions are modeled as a single feature without any child nodes and a string attribute that is defined by default or by user in the configuration tree. These features can be detected by the generator by checking a feature nodes ‘type’ attribute. This attribute is set to “STRING” as shown in the example XML export above for the COMPONENT feature. The

generator translates these features into a variable definition of the form “*name = value*”, using the features name as the variables name and the features string value as the value for the new variable. Each such variable definition and a new line character will be appended to the temporary output string “output”, that will be used for assembling the makefile.

3.3.2 Translating features representing boolean flags

Since there is no attribute of data type boolean in a features property tag, the ‘type’ value of that feature node is set to “NONE”. Boolean variables are modeled as optional features that are either activated or deactivated and do not have any attributes or child feature nodes. The XNP feature of the above XML export example represents such a boolean variable, how it is modeled and its XML export.

Intentionally, the generator should translate `SELECTED` config states into a ‘TRUE’ value assignments, `ELIMINATED` config states into ‘FALSE’ value assignments and skip `UNDECIDED` nodes, so these variables do not exists in the makefile. Unfortunately, nesC’s makefile mechanism uses boolean variables in different ways. Some conditions check whether a variable has the value ‘TRUE’ or not and some others check whether its ‘FALSE’ or not. This is the common way of using boolean variables. However, makerules also makes use of whether such a variable is declared and existent or not. Therefore a translation of `ELIMINATED` feature nodes into a ‘FALSE’ value assignment would specifically define this variable and mess up all conditions that are checking for variable existence. Rewriting all makerules conditions that use the existence criteria is one way of solving this issue, but since makerules does not make use of checking for an explicit ‘FALSE’ value it is easier to not translate `ELIMINATED` and `UNDECIDED` feature nodes into any variable assignment. These variables are not defined in the out-coming makefile and all the conditions will be executed correctly.

Feature nodes that have the ‘type’ value “NONE” do not have any attributes. Therefore they are either boolean variables or their value is defined in one or more of its child nodes. The generator checks whether the “NONE” type feature nodes has any child nodes and if it does not it is clear that this one is a boolean variable. In this case the generator checks the config state value of that

feature and produces a variable assignment statement of the form “*name* = TRUE” if the feature node is SELECTED.

3.3.3 Translating features with limited value choices

Variables like SENSORBOARD in the XML export example above can take a value only from a predefined set of possible values. In this example the value of SENSORBOARD can be either “basicsb” or “micasb”, but nothing else. These variables with a limited set of possible values are model as a feature that has a feature group as its child and this feature group has one feature for each possible value. The names of the child features give the string of one possible value. These child features do not have any attributes or further child nodes. The parent node has the name of the variable it represents and no attributes.

Because of that, the (data) ‘type’ value of the parent feature node is set to “NONE”, like boolean feature nodes. This time the node does have a child, in fact exactly one child (the feature group) and therefore it is not treated as a boolean variable. The generator then parses through all of the feature groups child nodes and reads the names of all SELECTED and mandatory features and appends all these names to one string. For definitions like SENSORBOARD, the feature group’s cardinality constraints are responsible that only one of the child features can be SELECTED at a time. Using the name of the parent node and the names of the selected child nodes gives the variable assignment statement that is appended to the temporary output. Since the name of the child feature nodes might not be capable to display some special characters the string attribute of such child features is used for creating the assignment statement, if defined. The feature model tree does not show a features string attribute directly, therefore it is more convenient to use feature names whenever possible, which are shown in the tree. We decided to support both methods in the generator, so it is not necessary to define string attributes, but if they are defined they will be used.

The top level feature node, called “makefile” in our example, does not have any attributes and it does not have a feature group as a child, either. We mentioned in the beginning, there is not cardinality constraint on the root node, that would restrict the amount of variables in makerules,

hence, there is no reason for having a feature group between the root node and the feature nodes that represent the variables. Therefore the generator parses child nodes of features whether there is a feature group between parent and child or not.

3.3.4 Building the makefile

Once the generator translated all features into variable assignment statements, they are stored in the temporary string variable “output”. The generator then takes this temporary output string and appends a modified version of the original makerules file to it. This gives the new makefile, that has all parameters set according to the feature model configuration. The original makerules file has to be modified, e.g. statements like default value assignments, that are now covered by the model should be removed.

3.4 Dependency Analysis

This section is intended to give an overview how to identify the important features in order to develop a software product line. Since makerules is written in the GNU make language it is capable for more things than just collecting values through parsing other files. In fact makerules uses some methods to get values from other places than just the other two makefiles. There are a few parameters depending on system environment variables like the users home directory and a few more depend on results of executed shell commands (e.g. for determining operating system information). Looking at the nesC makerules file, there are a lot of variables, dependencies and computations of values. Each one of those variables represents a point of choice, where a developer or the system needs to decide whether this parameter is important for the application and what value it should carry. The feature model we get from the makerules analysis has 33 such variables / features, some of them are related to the application (e.g. frequency settings or application name) and some of them are just for compilation purpose (like pathnames or operating system information). Furthermore there are a lot of inner dependencies among those variables, like a particular sensorboard can only be used if a certain platform was already chosen.

In order to build a product line for a particular nesC application (i.e. a mote within an application), an analysis of those variables, their purpose and their dependencies is needed. The following table divides the 57 variable mentioned in makerules in different groups according to their purpose and usage:

- 3 are special variables representing features of the make language
HOME, MAKECMDGOALS, MAKE
- 7 are simple string value assignments, to not use the same string multiple times while keeping them easily configurable
OBJCOPY, SET_ID, PROGRAMER, XNP_DIR, NCC, LIBS, KEYNAME
- 4 are temporary variables for extracting values from the command line
MIB5100, MIB510_, EPRB0, EPRB_
- 1 that defines a string for error outputs, how to build an application properly

USAGE

- 2 are used for recursive inclusion of makerules itself

MAKERULES, QUELL_RECURSIVE_TINYOS_APPS_MAKERULES

- 9 variables pointing to files or directories by using other variables

BUILDDIR, MAIN_EXE, MAIN_SREC, MAIN_TARGET, KEYFILE, BASEDIR, DOCDIR,
BOOTLOADER, PROGRAMMER

- 1 variable is the result of an executed shell command

TINYSEC_KEY

- 4 variables which are not modified nor used and just passed to the final compiler step

BUILDLESS_DEPS, BUILD_EXTRA_DEPS, CFLAGS, LDFLAGS

- 3 variables for debugging and documentation generation

DBG, DBGOPT, DOCS

- 2 variables for determining what are the possible target platforms

PLATFORMS, PLATAUX

- 2 variables can be completely computed by the system

PROGRAMMER_FLAGS, PROGRAMMER_FLAGS_INP

- 19 variables which might be set directly by the developer

DEFAULT_LOCAL_GROUP, OPTFLAGS, NESC_FLAGS, MSG_SIZE, PFLAGS, XNP,
COMPONENT, TINYSEC, PROGRAMMER_PART, PROGRAMMER_EXTRA_FLAGS,
MIB510, EPRB, AVRISP, AVRISP_DEV, PROGRAMMER_EXTRA_FLAGS_MIB,
PROGRAMMER_EXTRA_FLAGS_STK, PROGRAMMER_EXTRA_FLAGS_AVRISP,
SENSORBOARD, BOOTLOADER, PLATFORMS

Out of those 19 variables there are:

- 4 default value definitions

DEFAULT_LOCAL_GROUP, OPTFLAGS, NESC_FLAGS, SENSORBOARD

- 2 boolean representing flags

XNP (defined or not defined), TINYSEC (“true” or not defined)

- 12 read-only variables

MSG_SIZE, COMPONENT, PLATFORM (after initial definition), MIB510, EPRB, AVRISP, AVRISP_DEV, PROGRAMMER_EXTRA_FLAGS_MIB, PROGRAMMER_EXTRA_FLAGS_STK, PROGRAMMER_EXTRA_FLAGS_AVRISP, PROGRAMMER_EXTRA_FLAGS, PROGRAMMER_PART

- 1 variable that may have a predefined value but while processing makerules strings are added to it

PFLAGS

The idea is to simplify the first feature model by excluding product irrelevant properties. Therefore, variables like make language features or string constants are not included in the model, because they do not reflect any variability points of the application.

We start this analysis by looking at one of the most important features in makerules definition and see where it gets its information from. Each of these inputs need to be classified whether their values are user defined or computed by the system (e.g. environment variables or pathnames).

By extracting each occurrence of an assignment to PFLAGS we get the following dependency list where PFLAGS final value comes from:

- MSG_SIZE
- XNP
- XNP_DIR
- COMPONENT
- DEFAULT_LOCAL_GROUP
- PLATFORM
- BUILDDIR
- NESC_FLAGS
- TINYSEC
- TINYSEC_KEY

The PFLAGS variable is collecting a lot information from other values combining them in one string and passes it to the compiler. It depends directly on the ten variables above, whereas three of them (TINYSEC_KEY, XNP_DIR, BUILDDIR) are just pathnames or filenames. Since PFLAGs assignment statements are always of the form “PFLAGS := something \$PFLAGS” there is no initial value of PFLAGS. Hence we have either to add a “former PFLAGS” item to the dependency list or we have to highlight it in some way in the dependency diagram to see this item may has a user defined value. However, PFLAGS value itself is never used for any dependencies except the final compiler step. Therefore its value contains redundant information given by other variables, which makes it possible to compute the complete value of PFLAGS and so there is no need for having PFLAGS in our model except its initialization value.

The following picture shows PFLAGS and all its dependencies. Highlighted items are initially defined by user - non-highlighted items are computed (e.g. by a shell command) or their value depends on other items without being initialized.

Most items in this list are exclusively for computing PFLAGS but one of them (PLATFORM) affects a list of further items:

- BOOTLOADER
- SENSORBOARD
- MAIN_TARGET
- OPTFLAGS
- PROGRAMMER_PART
- PROGRAMMER_FLAGS_INP

BOOTLOADER is computed by the system and represents a pathname to a file. Its value depends on XNP_DIR which gives the directory and PLATFORM which decides if the file for mica2 or mica2dot should be used. If the value of PLATFORM is set to anything else BOOTLOADER will not be defined.

SENSORBOARD can have a user defined initial value, but if not a default value depending on the current value of PLATFORM will be assigned.

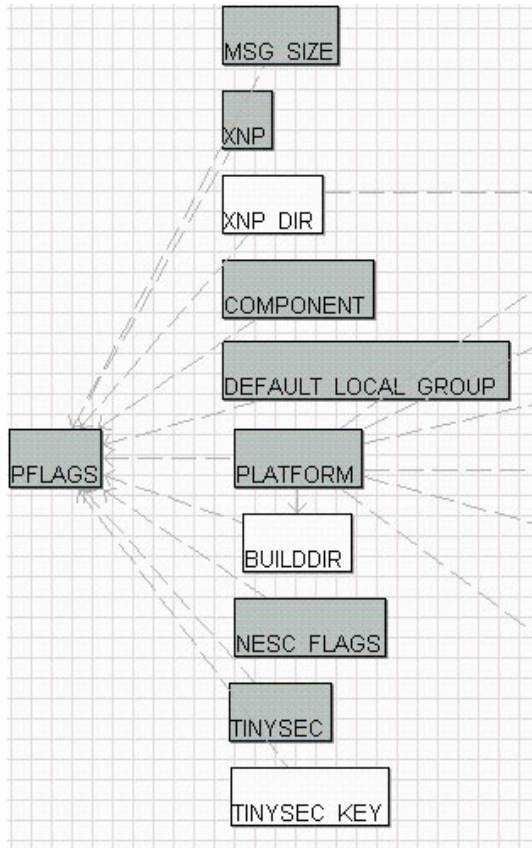


Figure 3.6: *Dependencies of PFLAGS*

MAIN_TARGET is computed by the system and depends if the value of PLATFORM is set to “pc” or not.

OPTFLAGS can have a user defined initial value, but if not a default value will be assigned. However depending on the current value of PLATFORM if it is set to “pc” OPTFLAGS former value will be overwritten. Furthermore OPTFLAGS depends on two more items “DBG” and “DBG_OPT”, both user defined exclusively. If DBG is set, it will overwrite OPTFLAGS again, and if DBG_OPT is set, it will add a parameter string no matter which part set OPTFLAGS value. The purpose of PROGRAMMER_PART is still unclear. There are several assignments to PROGRAMMER_FLAGS where PROGRAMMER_PART is involved to compute the value to be assigned. Since there is no former assignment to PROGRAMMER_PART it must have a user defined initial value. Confusing is that after all computations where it is involved the value of PROGRAM-

MER_PART is overwritten depending which PLATFORM is chosen, but after this assignment it is never used for anything.

PROGRAMMER_FLAGS_INP is computed by the system with several dependencies. First of all it depends on the choice of the value of PLATFORM. There are four sections for different platforms in the definition of PROGRAMMER_FLAGS_INP:

- mica
- mica128
- mica2
- mica2dot

Each of these has one more conditions depending on the value chosen programmer board. If programmer board “AVRISP” is chosen the value of PROGRAMMER_FLAGS_INP is set and it will include a parameter with the string representation of “AVRISP_DEV”. Otherwise if any other programmer board is chosen its value is set and it will include “PROGRAMMER_EXTRA_FLAGS”. Unfortunately there is no variable holding the chosen programmer board. There are four different programmer boards mentioned in makerules. If nothing is specified makerule will assume a parallel programmer board called “DAPA”. The three other choices are “MIB510”, “EPRB” and “AVRISP”, but each of them has an own boolean flag whether this board should be used or not. Unfortunately it is up to the developer to make sure only one programmer board is set. Furthermore each one of them has a user defined variable holding specific information to be added to the value of “PROGRAMMER_FLAGS”.

This issue can be solved by introducing two new items:

- PROGRAMMER_BOARD

This item intends to hold information which programmer board should be used. It replaces the former three variables “MIB510”, “EPRB” and “AVRISP”.

- PROGRAMMER_EXTRA_FLAGS_BOARD

This item intends to hold additional information for one specific programmer board. These information need to be added to PROGRAMMER_FLAGS if PROGRAMMER_BOARD is

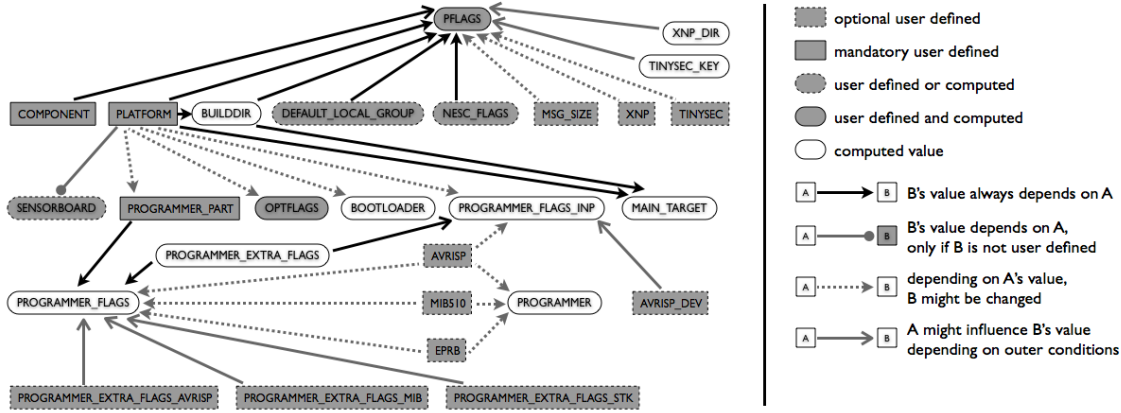


Figure 3.7: *Dependencies of makerules*

set to any non-default value. Since there can only be one programmer board at a time there is no need to have three different variables.

The following picture shows the refactored dependency diagram:

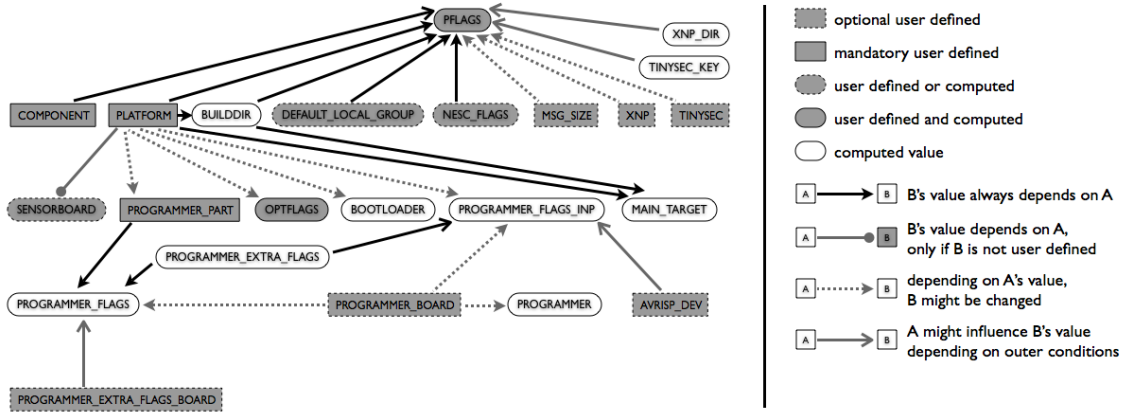


Figure 3.8: *Dependencies of makerules (refined)*

As an example how the staged concept of makefiles may affect the feature model, there are groups of application using a specific wireless device, that requires certain parameters. These parameters are appended to one string and passed through the makerule file directly to the compiler. There is no assignment or condition using this variable which is called “CFLAGS”. CFLAGS is used to hold information like radio frequency and transmission power. These values are defined in

another makefile which is imported to makerule.

In this file CFLAGS depends on four different properties:

- DCC2420_DEF_CHANNEL
- DRADIO_XMIT_POWER
- DCC1K_DEFAULT_FREQ
- DCC2420_TXPOWER

To each one of those parameters there are a lot of possible values in this file. All of them are commented out except one which is defining a part of CFLAGS. The following picture shows an example of a complete dependency diagram of makerule parameters including CFLAGS:

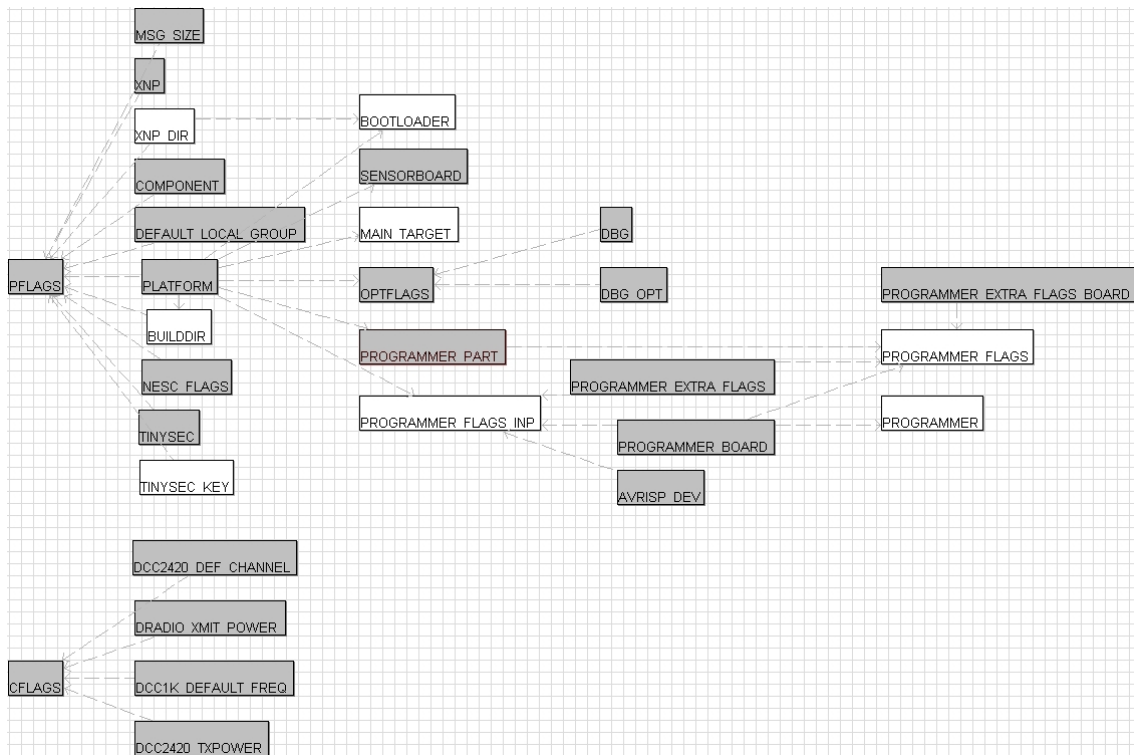


Figure 3.9: *Dependencies of makerules (refined) including CFLAGS*

All grey highlighted items need to be or can be defined by user. This small subset of makerule parameters gives enough information to generate a makefile but it is way easier to understand. All white colored items are calculated by the system at runtime. They do not need to be defined by

a user and should be hidden from the feature model.

Other than the initial value of PFLAGS everything that affects PFLAGS is computed automatically by makerules. Therefore we can also omit PFLAGS from the feature model because all these information would be redundant. We only need to include PFLAGS initial value if necessary and the makefile generator will append all computed PFLAGS parameters to this initial value.

Chapter 4

Software Product Line Configuration

We have seen, that the feature model of the previous section together with the generator we developed is capable to produce a makefile that configures all parameters for a specific build. However, the feature nodes of this model do not really reflect a software product line. These features are still closely mapped to makefile parameters, so a developer needs to know about the purpose of these features (i.e. makefile variables). We want to get a more abstract model, that deals with capabilities and hardware selections of a nesC application. For example, the developer wants to specify what types of sensors are needed to achieve the applications purpose, like light or motion sensors, rather than specifying necessary parameters for the selected hardware. Furthermore, if a certain kind of sensor (e.g. light sensor) is available from multiple vendors and each vendor requires to use a proprietary platform in order to use this sensor. This may rule out the use of other kinds of sensors, if they are not available from the same vendor or if they are not available for this platform. These compatibility constraints of hardware selections can not be specified using the previous feature model, since this model is limited to makefile parameters, that do not capture hardware selections.

A more abstract feature model is needed to make these selections and specify compatibility constraints. We introduce a high level of abstraction feature model by extending the previous model. The one-to-one mapped feature model has a root node called ‘makefile’, where the variability of makerules variables is specified. By a second root node, that takes care of high level

information ('high-level') and keeping the 'makefile' node in the model, we can extend the feature model. With a few changes of the generator to process only feature nodes of the 'makefile' subtree, the feature model tree may include information, that do not affect the makefile output, directly, such as a more abstract structure of hardware compatibilities. The feature model plugin allows constraints only within the same root node subtree. Therefore, it is necessary to have a common root node 'nesC node' with the child nodes 'high-level' and 'makefile'. We then use the feature model constraint mechanism on the new root node to define relationships between the high level subtree and the makefile parameter subtree.

The high level feature model focusses on a particular node of the sensor network application, its sensing and communication capabilities. Sensing capabilities depend on the hardware component (sensor board) selected for this node. While a software architect is usually concerned about the capabilities of a node, a developer is concerned about the hardware components and which platform, programmer board and sensor board to use. The following table list available hardware components and sensing capabilities of available sensor boards:

Platform	Programmer	Sensorboard	Sensorboard	Sensor Capability
mpr300	mib500	mts101	mts101	light, thermistor
mpr310	mib510	mts300	mts300	light, thermistor, sounder, microphone
mpr400	mib520	mts310	mts310	light, thermistor, sounder, microphone, accelerometer, magnetometer
mpr410	mib600	mts400	mts400	light, thermistor, accelerometer, pressure, humidity
mpr420	spb400	mts420	mts420	light, thermistor, accelerometer, pressure, humidity, gps
mpr500		mts510	mts510	light, microphone, accelerometer
mpr510		mda100	mda100	light, thermistor
mpr520		mda300	mda300	thermistor, humidity
mpr2400		mda320	mda320	no onboard sensors
tpr2420		mda500	mda500	no onboard sensors
xm2110		telosb		

Figure 4.1: *List of available hardware components and sensing capabilities*

Even if platforms and sensor boards have a specific product name, they also belong to a certain

product family, because they have the same connector or other things in common. The following table shows how they are grouped in platform families and sensor board families. These family names are used to specify the makerules variables, instead of the more specific product name.

Platform Family	Platform
mica	mpr300, mpr310
mica2	mpr400, mpr410, mpr420
mica2dot	mpr500, mpr510, mpr520
micaz	mpr2400
telosb	tpr2420
iris	xm2110

Sensor Family	Sensorboard
basicsb	mts101
micasb	mts300, mts310
mica2wb	mts400
mica2gpswb	mts420
sensorIB	mda300
mica2vibe	mda420
micawb	mep500
mica2dotsb	mts510
mica2dotgpb	mda500

Figure 4.2: *List of platform families and sensor board families*

A platform is the center part of a nesC mote, where a sensor board can be attached to it. Depending on the connectors of a platform it supports certain compatible sensor boards. Furthermore, each platform requires the use of a compatible programmer board to upload an application executable. The following tables list which platform family is compatible with which programmer boards and which sensor boards.

Programmer	Compatible Platforms
mib500	mica, mica2, mica2dot, micaz, iris
mib510	mica, mica2, mica2dot, micaz, iris
mib520	mica2, micaz, iris
mib600	mica2, mica2dot, micaz, iris
spb400	mica2, micaz, iris, telosb

Sensorboard	Compatible Platforms
mts101	mica, mica2, micaz, iris
mts300	mica, mica2, micaz, iris
mts310	mica, mica2, micaz, iris
mts400	mica2, micaz, iris
mts420	mica2, micaz, iris
mts510	mica2dot
mda100	mica2, micaz, iris
mda300	mica2, micaz, iris
mda320	mica2, micaz, iris
mda500	mica2dot

Figure 4.3: *List of hardware component compatibilities*

With all these information we can develop a high-level feature model that covers available hardware components as well as its sensing capabilities. Using the feature model constraint mechanism

we can also encode all the above hardware compatibilities. The feature model tree can be organized quite arbitrarily, since the generator does not parse the high-level subtree. The high-level model focusses on the three main hardware components ‘platform’, ‘sensors’ and ‘programmer’, where each one of them is modeled as a feature. Sensing capabilities, for example, are organized under the ‘sensors’ feature node as well as a feature group for specific sensor boards, see figure 4.4(d).

Additional constraints, that define the relationships between the high-level subtree and the makefile subtree are necessary. These constraints enforce, that makefile parameters are set correctly, once a selection in the high-level subtree is specific enough to define a certain makefile variable. For example, if the developer chooses the “mpr300” platform, that belongs to the “mica” family, the PLATFORM variable is set to ‘mica’ by selecting the ‘mica’ feature of the PLATFORM’s feature group in the makefile subtree. The generator will then parse the makefile subtree and produces a correct and valid makefile to compile the application.

Capturing constraints

Figure 4.4(c) shows an FMP constraint stating that the ‘mib520’ programmer board is only compatible with the ‘mica2’, ‘micaz’, and ‘iris’ platform families. Figure 4.4(d) demonstrates the automatic constraint checking of the FMP tool; when the ‘sounder’ sensing capability is chosen, all sensing boards except ‘mts300’ and ‘mts310’ are automatically eliminated from consideration (indicated by the fact that the board selections are gray-ed out and have an “x” (as opposed to a check-mark) in the corresponding selection box).

Figure 4.5 summarizes the types of constraints and number in each category that we currently have coded in the tool. The constraints can be categorized as hardware compatibility constraints, taxonomy/capability constraints, and dependence constraints. The concept of hardware compatibility constraints is reasonably clear; we have mined this information from vendor documentation. Constraints between libraries and hardware are often unanticipated. For example, an MDA100 can coexist on the same Mica-class mote as an MTS420 (if the developer wanted a thermistor/humidity sensor that knew its own location via GPS), but

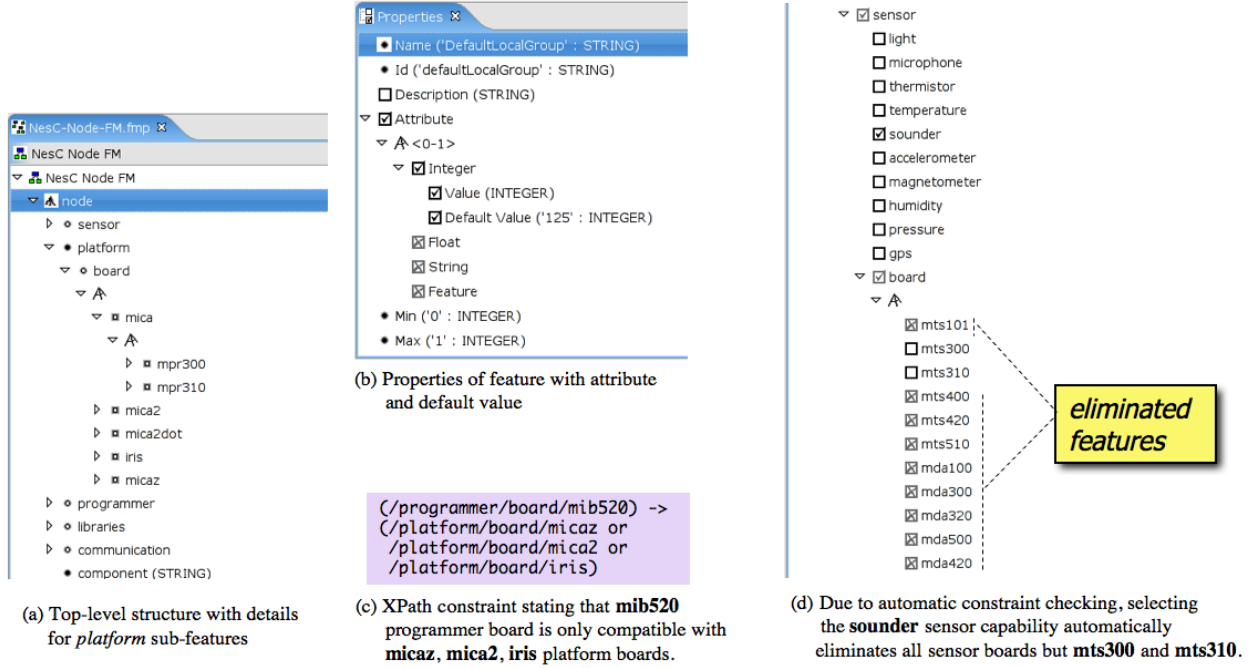


Figure 4.4: FMP tool screenshots

#	Constraint Category
10	Sensor Board \Leftrightarrow Platform Family Compatibility
5	Programmer Board \Leftrightarrow Platform Family Compatibility
10	Programmer Board \Leftrightarrow Connection Capability
30	Sensor Board \Leftrightarrow Sensing Capability
32	Platform Board \Leftrightarrow Platform Capability
	Library \Leftrightarrow Hardware Compatibility
	InterLibrary Dependence

Figure 4.5: Summary of constraint categories

current software libraries do not support combining an MTS420 and MDA300. As another example, the documentation for the CC2420Radio library states that it only works with MicaZ and Telos platforms. We have not attempted to make an exhaustive specification of constraints associated with libraries (thus, it is not particularly meaningful to report the number of constraints for those categories in Figure 4.5), but our experience so far indicates that it would be very valuable to have a framework like this that would allow nesC contributors to formally document constraints in a configuration tool that guarantees adherence to those constraints as the system is being designed.

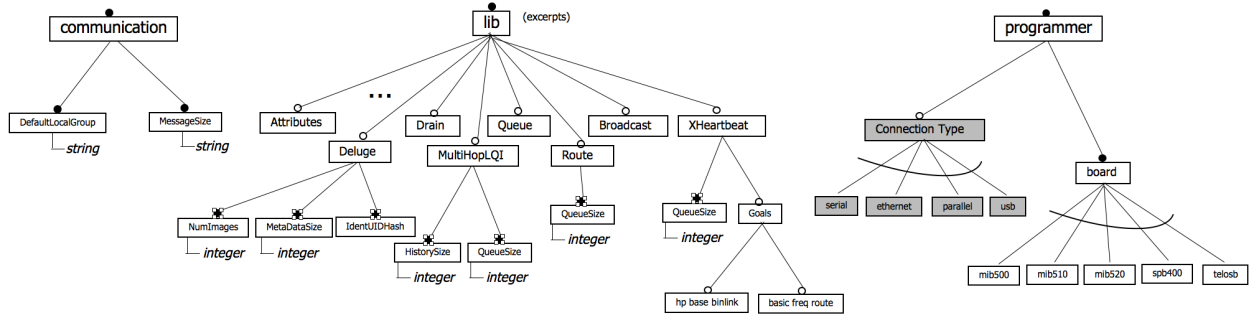


Figure 4.6: *Additional feature model excerpts for nesC node configuration*

Capabilities and top-down design

We believe the ability to specify constraints and incrementally solve them as design unfolds (via feature model tools) can have a very powerful impact on sensor network design and development. In following sections, we will discuss how this impacts team development and software product line design for sensor networks. A more direct impact can be seen in our ability to move beyond the “bottom up” approach to configuration forced by the makefile paradigm (design from available hardware components, libraries) to support more “top down” design approaches (design by identifying basic functionality/capabilities needed in the sensing domain). As a simple illustration of the potential, gray feature nodes in Figures 2.5 and 4.6 move beyond choice points present in the makefile structure to offer selections based on capabilities of sensor boards (light, GPS, thermistor, humidity, etc.) platforms (communication frequency, etc.) and programmer boards (usb, serial, ethernet, etc. connections). The constraint mechanism formalizes the relationship between capabilities and hardware components, and the FMP tool support guarantees that selection of desired capabilities automatically exposes to the developer the impact of the selection (the selection possibilities for available hardware are narrowed). The infrastructure that we have developed can be enhanced with a variety of different taxonomies.

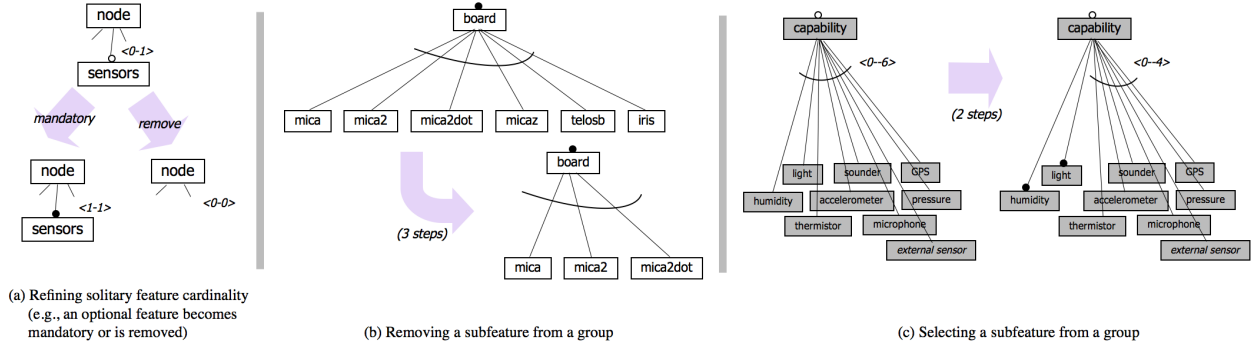


Figure 4.7: *Example feature model specialization steps*

Staged selection of features

Czarnecki *et al.*⁸ provide a formal semantics via context-free grammar for the incremental specialization of feature models, where specialization is phrased in terms of a collection of specialization operators. We give a brief overview of the relevance of some of these operators to our context.

Refining feature cardinalities:

Solitary features may have cardinalities; for example $\langle 0-1 \rangle$ indicates that a feature such as ‘sensors’ in Figure 2.5 is optional. A cardinality may be refined in a specialization step by narrowing the interval of the cardinality. For example, $\langle 0-1 \rangle$ for ‘sensors’ can be refined to $\langle 1-1 \rangle$ indicating that sensors will be included in the functionality, or $\langle 0-0 \rangle$ indicating that sensors will not be included (Figure 4.7(a)).

Removing a subfeature from a group:

A specialization step can alter a feature group by *removing* a feature from the group as long as cardinalities permit. For example, in the ‘platform board’ feature group of Figure 2.5, the subfeatures ‘micaz’, ‘telosb’, and ‘iris’ may be removed in a sequence of steps to indicate that boards from those families will be not used. However, it would be invalid to remove all the board family subfeatures because this would violate that feature group cardinality (expressed by the arc across the lines to the children) that one of the subfeatures must be chosen (Figure 4.7(b)).

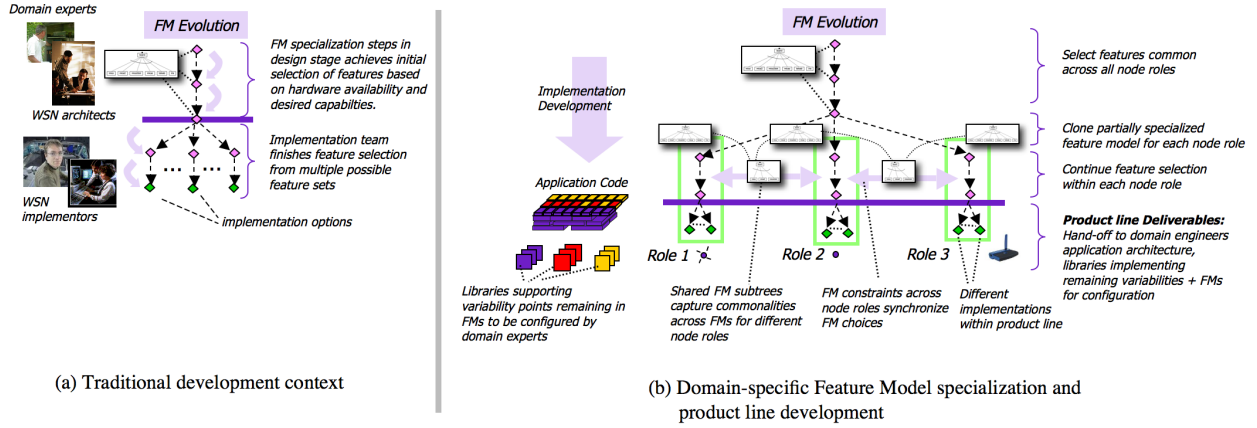


Figure 4.8: *Feature-model-orchestrated development scenarios*

Selecting a subfeature from a group:

A specialization step can alter a feature group by *selecting* one of the features from the group as long as cardinalities permit. For example, in the ‘sensors capability’ feature group of Figure 2.5, the subfeatures ‘light’ and ‘humidity’ could be selected in a two step sequence to indicate that any sensor board chosen for the node must support both light and humidity sensing. Note that such “choice” steps cause the cardinality of the feature group to be adjusted to reflect the choices (Figure 4.7(c)).

Other specialization steps include the assigning of an attribute value to a feature (which in our case typically corresponds to setting a parameter value for a library, selecting a communication frequency, etc.), and cloning of a solitary feature (when cardinalities permit).

Figure 4.8(a) illustrates one scenario in which the steps above would be applied in a traditional development context: system architects gather requirements from domain experts and use FM specialization steps to make initial selection of features. Then, the partially specialized feature model is handed off to implementation teams. The initial feature selection presented by the architects guides and constraints the implementation teams as they choose among implementation options and move toward the completed executable.

CBNF feature models also allow *sharing* of sub-trees of a feature model which allows selections in one tree to be reflected in another tree. In our setting, this can be used to e.g.,

share communication settings across feature models for multiple nodes as illustrated in Figure 4.8(b); we will discuss this in greater detail below.

Compiling to makefiles

We have developed a compiler that translates our feature models to makefiles that work with the existing nesC GlobalMakerules makefile structure. It should be clear that the feature information from our model excerpts in Figures 2.5 and 4.6 is sufficient for generating values for most of the variables highlighted in Figure 3.7. Values for the other variables such as debugging options, compilation infrastructure pathnames, and other flags come from other sections of the feature model not included in the figures or from other configuration settings (e.g., those that pertain only to compiler settings).

Chapter 5

Evaluation

5.1 Examples

With this new developed feature model and the makefile generator, we want to show how to configure a few example applications, to point out strengths, weaknesses and work-arounds. We use example applications that come with the nesC distribution and show how their makefiles can be translated into a feature model and what the generator produces using this feature model.

Blink

The first example is a very simple application called “Blink”. The application makefile of Blink has only two lines of code:

```
COMPONENT = Blink
include ../Makerules
```

The second line calls the global makerules file and is not a parameter of the Blink application. A local makefile is not defined, so there is only the mandatory parameter “COMPONENT” configured in Blink’s makefile. The only feature we can derive from this parameter is the COMPONENT feature in the makefile subtree (s. figure 5.1).

The generated makefile combines information from the original application makefile with the original global makerules file. A compilation of either one approach will produce the exact



Figure 5.1: *Blink's configuration tree and makefile output*

same output.

The only question is: Where should we put the COMPONENT feature in the model tree? On the one hand, naming an application is not really something special, where a developer needs to know about all deep application parameters, so COMPONENT should be in the high-level subtree. On the other hand COMPONENT is a parameter that has to be set in the makefile, so it does belong to the makefile subtree. In fact we need to have it in the makefile subtree, so the generator parses it and includes COMPONENT in the output makefile.

It would be nice to have COMPONENT modeled in the high-level subtree and the generator accesses it in some way. One solution could be to have it modeled in both sub-trees, but this would cause redundant information and since the current feature model plugin and its constraint mechanism does not support checking and comparing string attribute values of features, it could cause inconsistency.

To have the capability of referencing features would be very helpful. The current feature model plugin has something that is called reference, but this is rather a copy/paste mechanism than a reference. Once a reference is 'unfolded' it inserts a copy of the referenced

feature and its sub-tree, but changes of the original feature do not affect unfolded references. There should be a reference that keeps the relation to the original feature and allows modification either way together with the capability of renaming features . This mechanism would be helpful to organize the feature model tree without causing redundant information. The COMPONENT feature, for example, could be modeled in the makefile sub-tree and ‘link’-feature named “NAME” would represent COMPONENT in the high-level sub-tree.

MicaHWVerify

```
PLATFORMS = mica mica2 mica2dot
COMPONENT = MicaHWVerify
include ../Makerules
```

In this example we have one more line that specifies a set of possible platforms. However, our feature model does not cover this parameter, even in the makefile sub-tree. The PLATFORMS parameters is a predefinition of possible platforms before the decision which platform to be used for a build is made. Therefore, the model does not specify a platform, but we can rule out a few (s. figure 5.2).

The PLATFORM feature is modeled as a mandatory feature, therefore a configuration that does not specify a platform is not valid. Even if we rule out a few possible choices in this configuration, we did not specify a platform, so the cardinality constraint of the PLATFORM feature group is not satisfied. Unfortunately, the existing feature model plugin does not show whether cardinality constraints are satisfied or not and for additional constraint neither. If additional constraints are not satisfied, the generator would not start, but unfortunately there is no error notification.

A newer version of the feature modeling plugin has been developed. This 0.7.0 version is available at the website of the Generative Software Development Lab of University of Waterloo¹⁹, but it is not released as a stable version, yet. This version includes some major



Figure 5.2: *MicaHWVerify's configuration tree and makefile output*

improvements of the constraint mechanism. Cardinality and additional constraints are listed in a tree structure and each constraint is marked whether it is satisfied or not. Therefore, the issue described earlier is solved once the new version is released.

HighFrequencySampling

```

PLATFORMS = mica mica2

COMPONENT = HFS

SENSORBOARD = micasb

include ../Makerules

```

In this example we have a set of possible platforms and a sensorboard specified. Like in the previous example, PLATFORMS is not a specification, but an elimination of certain features. However, the SENSORBOARD feature is a specification of a value and therefore the generator produces an output that includes a SENSORBOARD definition(s. figure 5.3).



Figure 5.3: *HighFrequencySampling*'s configuration tree and makefile output

SenseToRfm

```
COMPONENT=SenseToRfm
PFLAGS=-I%T/lib/Counters
include ../Makerules
```

In this example we have a name (COMPONENT) and a library to be used for compilation. Libraries are specified as part of the PFLAGS variable. Since most of PFLAGS values are computed we omitted PFLAGS from being modeled in the makefile sub-tree. The above definition is not computed automatically and has to be modeled. We add a new feature called PFLAGS to the makefile sub-tree and define a string attribute “-I%T/lib/Counters”, because this definition has only one value.

Furthermore, we can model libraries in the high-level sub-tree by adding a feature group called “libraries” and a new child feature for each library. Unfortunately, this causes redundant information and additional constraints have to be specified to ensure consistency of high-level and makefile sub-tree. (s. figure 5.4).

The definition statement of PFLAGS add the specified value to the former value of PFLAGS. This behavior was adopted from the way PFLAGS is used in makerules and implemented in the generator for the first feature model. A feature with the name “PFLAGS” is a special case in the implementation of the generator. It adds “\$PFLAGS” to the end of the generated variable definition statement.

5.2 Domain-specific feature models

One of the goals of the feature modeling framework is to support domain experts belonging to an organization *Org* in developing sensor applications for a specific class, *SN_class*, of problems. As Figure 4.8(b) illustrates, in our framework, we analyze the requirements of *SN_class* to arrive at a general architecture for sensor applications in this family. This architecture is arrived at by starting from our initial feature model and applying specialization steps to select features (*e.g.*,

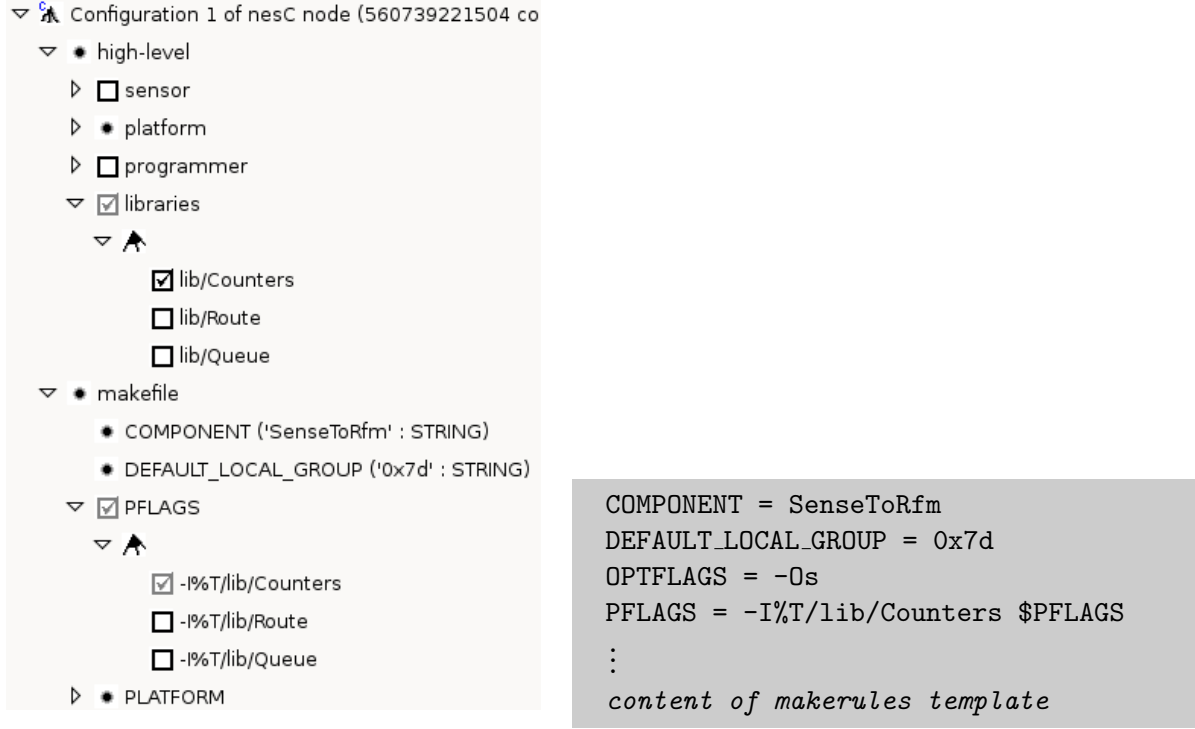


Figure 5.4: *HighFrequencySampling's* configuration tree and makefile output

hardware components, capabilities) that are common across all product instances and across all nodes in the application family. Next, we identify the various roles of the played by the motes in this architecture. We arrive at feature models for each of these roles by first cloning from the partially specialized common model (a cloning operation is supported in the operations of⁸) and then by specializing each role-specific model to arrive at specialized feature models via incremental feature selection. The requirements and the hardware resources available in *Org* will enable us to pre-select some of the options exposed in the general FM. Using this selection process, for each role r , we provide a feature model, FM_r , which is a partially instantiated model of our general FM. Any commonalities between the feature models for the various roles can be captured by sharing feature model tree nodes across the feature models for each role. In addition, constraints between feature models synchronize choices across roles.¹ Each FM_r may still allow the domain experts to exercise

¹Sharing constraints between role feature models is conceptually possible, but currently not supported in the FMP tool; we currently duplicate constraints by running some shell scripts to re-enter them in the tool for each node model.

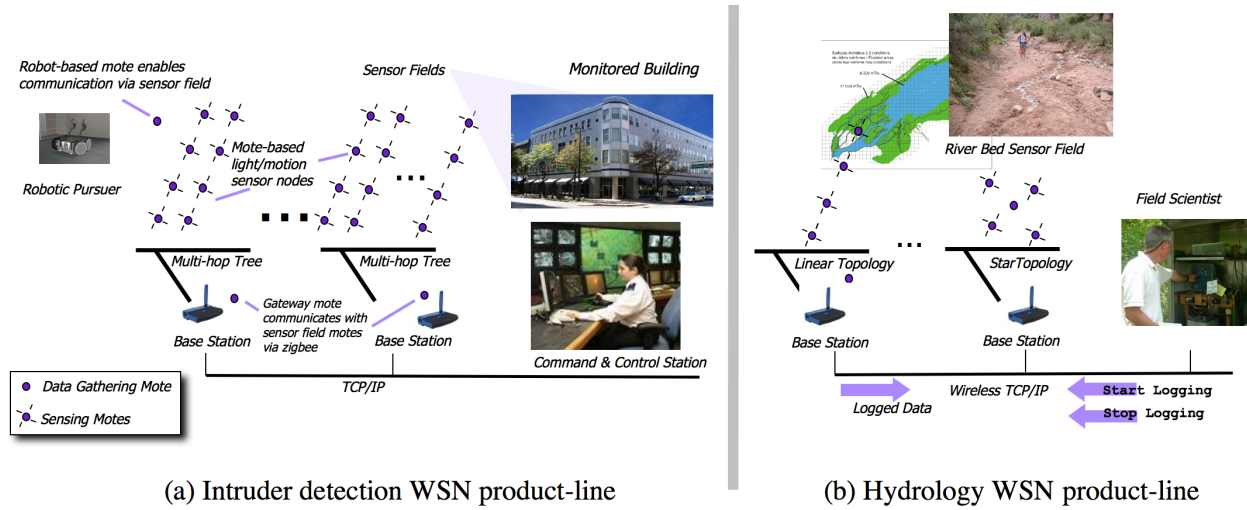


Figure 5.5: *Architecture for Intruder Detection and Hydrology WSN product lines*

a different set of choices. The framework requires that libraries to support these variabilities points be provided to the domain experts. Furthermore, for the class of applications whose functionality is known in advance, we provide application code to the domain experts (thus, selecting available options in the specialized FM is the only activity required to build an application). Alternatively, the domain experts may write the application code and use the delivered FMs and libraries to assemble and build the executables.

5.2.1 Intruder Detection

This section discusses the design of a family of applications to detect and track an intruder using light sensors in an area which may be sub-divided by natural boundaries (e.g., multiple rooms in a building). Figure 5.5(a) shows the architecture of a typical intruder-detection application in this family. The sensing field in each sub-area (grid) contains a number of motes with attached sensor boards and a base-station hosting a sensorless mote. Motes for each grid are assigned a group-id distinct to that grid. Each grid's sensing motes and the base station form a multihop tree rooted at the base station. The base station is connected via TCP/IP (wireless) to the control station. When a mote in grid G detects an intruder, the control station is notified via the mote's base station. The control station in turn directs a robot (pursuer) with an on-board mote to grid G

(referred to as the active grid). When the robot enters G , the pursuer may no longer have a WiFi connection to the control station. Therefore, it changes the group-id of its on-board mote to match the group-id of G in order to communicate to the control station. The control station then directs the pursuer to the last known position of the intruder via G' base station. If the intruder moves to another grid, the robot is directed to the new active grid by the control station.

There are three executables to be configured depending on the role played by the mote: sensor mote, gateway mote, and pursuer mote. Figure 5.6 displays the specialized feature model for the sensor mote derived from the model in Figure 2.5. The specialized feature model allows platform

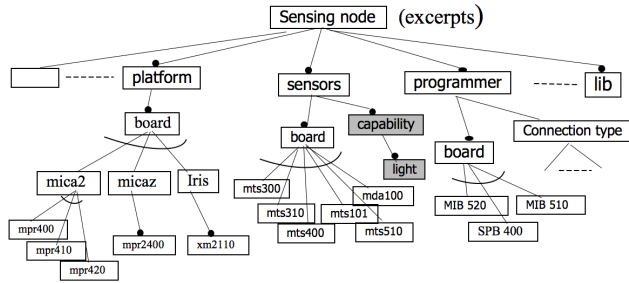


Figure 5.6: *Specialized feature model for intruder detection sensor mote*

boards from the Mica2, MicaZ, and Iris families (the choice was mainly limited by the hardware availability). Choosing the sensing capability of light narrowed the choice to seven available sensor boards. To implement the sensor grid routing, the gateway and sensor motes use the *XMesh* library from CrossBow. We pre-selected with extended low power (ELP) option for XMesh for the sensing node FM (not shown). The feature model for the gateway mote provides a similar choice for platform boards but has no selection point for sensor board. The moteid variable is set to 0 and the XMesh settings are left to the domain expert (which is set depending on how close the gateway is to the sensor grid). A number of commonalities were identified between the FMs of the various motes. For instance, each mote in a grid must have the same group-id. All platforms belonging to the same grid must use the same platform board (to be able to communicate). However, since we want the robot mote to be able to communicate with any of the grids, this further forced the same wireless module to be used in all of the grids. The sensorless mote of the gateway, whose task is mainly to forward messages, must also be able to communicate with the motes in its grid

(it does not have to belong to the same family as the grid motes but must communicate at the same frequency). The designer is free to select different sensor boards for different motes in the same grid. The FMP tool currently does not provide all the necessary support to establish the *inter-node* constraints described above. They can be achieved to some degree, e.g., by sharing the feature model subtree for ‘platform’ across all the nodes.

Domain experts implemented an application belonging to this family with MicaZ as the platform, MTS300 as the sensor board, and the Stargate as the base station. The robot mote was mounted on a Pioneer robot which played the role of the pursuer.

5.2.2 Hydrology – Ground Water Monitoring

This section discusses a very domain-specific class of applications for hydrological monitoring developed for a local research group. Applications in this class involve monitoring the quality of rural ground water runoff into streams, e.g., to assess level of chemicals in the water due to fertilizer use in surrounding farm land. The sensors are to be arranged in a linear configuration across a stream bed, which, given the size of some beds, can require a significant number of sensors. Also, to prevent potential damage to the gateway from flooding, it was to be situated up to couple of hundred feet from the end of the sensor array. The application (see Figure 5.5(b)) begins when a field engineer via a laptop client equipped with WiFi capabilities instructs a gateway to commence sensing activities. The gateway node sends a “start logging” command to a field of sensing nodes deployed on a river bed to start logging data readings of water properties. Each sensing node also has to be able to collect a water sample if a surge of water is detected. A pressure transducer that sunctions water into a plastic tube for later chemical analysis(similar to a small plunger) was designed for this task. Each sensor continues collecting data readings its onboard memory is full or until it receives a “stop logging” message from the gateway. Data from each sensor is sent to the gateway, which in turn forwards this data onto the client side where the data messages are parsed and interpreted.

For this class of applications, there are nesC executables needed for two roles, data gathering mote and sensing mote. Accordingly, two specialized FMs are needed. Figure 5.7 shows the

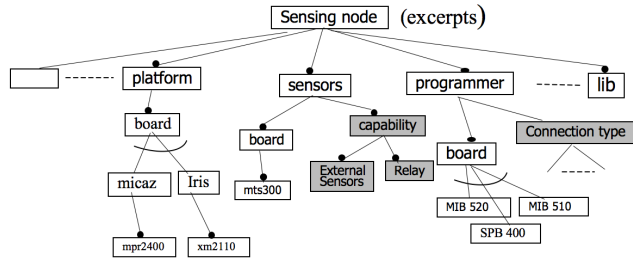


Figure 5.7: *Specialized feature model for hydrology sensor mote*

specialized FM for the sensing motes. After selecting the capabilities of transmission range and data rate for the platform node in the general FM (not shown), and looking at the available hardware, the specilized FM offered the choices of MicaZ and Iris. The capabilities required under the sensor node in the general FM were the ability to attach specialized external sensors and a relay (to drive the external pressure transducer). The constraints in the general FM restricted the choice to MDA300 only. For the data gathering mote (FM omitted), the additional requirement for the platform is Stargate compability. In this case also, the specialized feature model allows either MicaZ or Iris platforms. No sensorboard is required for the data gathering mote.

A number of additional challenges were addressed in developing the application code for this class of applications. The code was designed to interact with the sensorboard to adjust the sampling rate of the channels to maximize battery life. The relay to activate the transducer is driven by results from other sensed data (e.g., to detect a water surge).

5.3 Assessment

During our analysis of the nesC makerules file we found some more points of the existing feature model plugin that could benefit from improvements. We want to assess where the existing framework has weaknesses or where it is not rich enough to cover properties or constraints of the nesC makefile structure.

default value

As mentioned under point makerules in chapter 3.1 it is possible to define default variable declarations using the feature model plugin. In the properties section of any feature there are attributes which can be defined to have a numeric or string value. Once the data type is chosen there are two entries: “Value” and “Default Value”. The default value is specified when modeling the feature tree and if a user doesn’t specify its value in the configuration of the model the default value is used. At the time the configuration is created the model copies the default value to the normal value and automatically activates the normal value. In this framework a user can change the default value in the configuration step which leads to different default values for any new created nested configuration. Furthermore the user is able to deactivate the normal value, but from that time on it will never be automatically enabled anymore. This behavior doesn’t reflect the usual behavior of default value definitions in our makefiles. It is more a default value for further configurations. Modeling and configuration area should be separated from each other since there are different persons/roles who are responsible for changing entries in each area.

removing features with constraints

We have seen all features can have additional constraints associating two or more feature choices. Suppose there is a constraint defining that if feature A is activated feature B is automatically activated as well and vice versa. Specifying a certain configuration could also be done by removing features, which are not allowed in this specific branch and all its nested configurations. Removing feature A from a configuration doesn’t really remove the feature

but hides it from the display and from the XML export. A constraint which states that feature B cannot be activated without feature A will not be removed. Activating feature B will also enable feature A, which is now hidden and other constraints involving A will be computed. The conclusion is removing such a feature with an associated constraint is not the same like deactivating this feature. The removed feature can take any value and associated features might be selected arbitrarily or represent unreal condition.

accessing strings from constraints

It would be helpful to have a constraint language which is capable to check string values of features instead of just checking whether a feature is activated or deactivated. For example the “Component” variable is a freely editable string that would belong to the high level subtree as well as the makefile subtree, but it is not possible to force one feature to have the same string than another feature.

default choices

The FM plugin can specify default string attributes for features, but unfortunately there is no way to preselect and activate certain features by default. The problem is there might be a situation where a valid configuration is only reached if certain features are activated (e.g. feature A or feature B has to be enabled). A new created configuration has no features enabled and therefore it is in an inconsistent state. Defining additional constraints doesn’t help, because the constraint are not computed when a configuration is created. They are computed whenever the first selection is made. Furthermore the underlying constraint mechanism is not able to handle undefined values. The feature model uses three different states for features “enabled”, “disabled” and “undefined” but the constraint language can handle only two of those “enabled” and “disabled”.

Chapter 6

Related Work

Many feature modeling notations have been proposed. Besides the CBFM model⁸ used here, the Orthogonal Variability Model (OVM) of Pohl *et al.* is a popular notation²⁰, but little tool support for it exists and the details of incremental specialization have not been specified for it. Other notable approaches include the propositional logic-based notation of Batory *et al.*³.

Several taxonomies^{15,22} for aiding the design of WSNs. While our feature model framework can be viewed as providing a simple taxonomy of node-level features, the main goal of our work is to provide a formal model of features such that the selection of those features can be used to automatically configure executables – an issue which the work cited above does not address.

Schröder-Preikschat *et al.*²¹ have also recently proposed the use of feature models for configuring WSNs. However, their short paper²¹ does not present an actual WSN feature model, nor does it describe any tool support implemented. The ontology presented by Jurdak *et al.*¹⁶ is similar to our organization of features, but it does not aim to support the direct configuration of executable nor the type of product line specification that we require. Lee *et al.*¹⁷ describe a product-line approach for WSN development, but their focus is on code-level organization issues rather than use high-level modeling techniques.

A number of high-level abstractions have been proposed to simplify the design of sensor applications^{23,24}, and taxonomies to guide users in selecting the appropriate abstractions according to their programming approach (e.g., macro-programming vs node-level programming) have been

done¹³. Studies have also been done to identify the challenges arising in various network layers in different deployment scenarios and the appropriate protocols to address them².¹⁴ presents an evaluation a set of data gathering protocols to determine the application scenarios where each protocol outperforms the others. While these studies help in designing applications, they do not address issues such as those related to configuration option compatibilities in building executables for a node.

A number of TinyOS-based packages have been developed for specific classes of applications^{1,18,25} which provide application-level configurability to their users. For example, both TinyDB¹⁸ and Moteview¹ provide a graphics interface using which a user can select the attributes to be sensed (along with associated parameters such as sampling rate and duration), and view the results on a GUI. In these packages, the application executable is built to support multiple functionalities (alternatives for the same choice point); the dynamic selections of attributes allows the user to enable specific functionalities. The goal of our feature modeling framework, on the other hand, is to build executables supporting a specific combination of features.

Chapter 7

Conclusion

We have seen it is possible to use the notation of feature models to express the configurability of nesC-based wireless sensor network applications. We developed a feature model using the feature modeling plugin and a makefile generator, that parses a model configuration and generates a makefile that includes all variable settings from the model configuration. The biggest advantage of such a framework is to have the total configurability available in one graphical tree and the configuration of one particular application is based on the same global model tree. The only problem is to get the total configurability. Our model is based on example applications and their makefiles. Variables and parameters used in this makefiles are modeled, but there might be other parameters possible. Unfortunately, there is no documentation about available parameters, and values often depend on hardware selections. The CFLAGS variable, for example, is not processed by any makefile and passed directly to the compiler, therefore it is hard to find out the total configurability for CFLAGS.

Once this is done the model shows all options and a developer can configure an application using the high-level or the makefile-level sub-tree. It is easy to see that there are redundant information in the model tree (e.g. a sensorboard can be selected in the high-level and makefile sub-tree). To keep the model consistent we need to specify lots of constraints. It would be helpful to have link-features, in addition to the reference mechanism, so one feature that keeps the information can be accessed from multiple places using different names. With this mechanism, we could organize

all real variables in the makefile tree and organize the high-level tree quite arbitrarily. Currently, the COMPONENT feature, for example, has to be set in the makefile sub-tree only.

Another way of solving the COMPONENT feature problem is to use a constraint mechanism that is more powerful. Especially feature attributes need to be accessed (read and write) from within constraints. With this additional power, we could have two features for COMPONENT (one in each level of abstraction) and whenever a string value is specified for one of them, a constraint would update and synchronize the other feature automatically. This is almost the same behavior like link features, but link features would be easier to handle.

A more powerful constraint mechanism is also needed if we want to specify a model that handles multiple nodes of a network. First it is necessary to have multiple instances of the same feature and to organize these features according to the application design. This capability could be achieved by link features or the existing references with cardinality constraints (stating 1..n instances of a feature are possible). But using the current constraint mechanism it is not possible to specify constraints that for all instances/copies of a feature. The constraint language uses the identifier of one feature, but each instance/copy of this feature has its own unique identifier and therefore no constraints.

With all the improvements above, this feature model framework could configure complete applications with multiple nodes in a very comfortable way and the framework would gain lots of new applications and possibilities.

Bibliography

- [1] Moteview web site, <http://www.xbow.com/Technology/UserInterface.aspx>.
- [2] I.F. Akyildiz, T. Melodia, and K.R. Chowdhury, *A survey on wireless multimedia sensor networks*, Computer Networks (Elsevier) Journal **51** (2007), no. 4.
- [3] D. Batory, *Feature models, grammars, and propositional formulas*, Proceedings of the International Conference on Software Product Lines, October 2005, LNCS 3714, pp. 7–20.
- [4] S. Bühne, K. Lauenroth, and K. Pohl, *Modelling requirements variability across product lines*, Proceedings of the International Conference on Requirements Engineering, August 2005, pp. 41–50.
- [5] World Wide Web Consortium, *Xml path language (xpath)*, <http://www.w3.org/TR/xpath.html>, version 1.0.
- [6] Atmel Corporation, *AVR STK500 starter kit and development system*, http://www.atmel.com/dyn/resources/prod_documents/doc1939.pdf.
- [7] K. Czarnecki and U. Eisenecker, *Generative programming: Methods, tools, and applications*, Addison Wesley, 2000.
- [8] K. Czarnecki, S. Helson, and U. Eisenecker, *Formalizing cardinality-based feature models and their specialization*, Software Process : Improvement and Practice **10** (2005), no. 1, 7–29.
- [9] Free Software Foundation, <http://www.gnu.org/software/make/manual/make.html>, *The gnu make manual*, 0.70 ed., 2006.
- [10] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler, *The nesC language: A holistic approach to networked embedded systems*, Proceedings of the

- ACM SIGPLAN 2003 conference on Programming language design and implementation, ACM Press, 2003, pp. 1–11.
- [11] O. Gnawali, B. Greenstein, K. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler, *The tenet architecture for tiered sensor networks*, Proc. of ACM SenSys, 2006.
 - [12] Hassan Gomaa, *Designing software product lines with uml : From use cases to pattern-based software architectures*, Addison Wesley, 2004.
 - [13] R. Gummadi, O. Gnawali, and R. Govindan, *Macro-programming wireless sensor networks using kairos*, Proc. of the International Conference on Distributed Computing in Sensor Systems, 2005.
 - [14] J. Heidemann, F. Silva, and D. Estrin, *Matching data dissemination algorithms to application requirements*, Proc. of ACM SenSys, 2003.
 - [15] Aravind Iyer, Sunil S. Kulkarni, Vivek Mhatre, and Catherine P. Rosenberg, *A taxonomy-based approach to design large-scale sensor networks*, Wireless Sensor Networks and Applications (Yingshu Li, My Thai, and Weili Wu, eds.), Springer, 2008, pp. 3–33.
 - [16] R. Jurdak, C. Lopes, and P. Baldi, *A framework for modeling sensor networks*, Proceedings of the OOPSLA 2004 Workshop on Building Software for Pervasive Computing, 2004.
 - [17] Woojin Lee, Sungwon Kang, and Dan Hyung Lee, *Product line approach to role-based middleware development for ubiquitous sensor network*, CIT '07: Proceedings of the 7th IEEE International Conference on Computer and Information Technology (Washington, DC, USA), IEEE Computer Society, 2007, pp. 1032–1037.
 - [18] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, *Tinydb: An acquisitional query processing system for sensor networks*, ACM Transactions on Database Systems **30** (2005), no. 1.
 - [19] University of Waterloo Generative Software Development Labs, *Feature modeling plugin*, <http://gsd.uwaterloo.ca/projects/fmp-plugin/>, version 0.6.6.

- [20] K. Pohl, Günter Böckle, and Frank van der Linden, *Software product line engineering*, Springer, Berlin, 2005.
- [21] Wolfgang Schröder-Preikschat, Rüdiger Kapitza, Jürgen Kleinöder, Meik Felser, Katja Karneier, Thomas Halva Labella, and Falko Dressler, *Robust and efficient software management in sensor networks*, Proceedings of the 2nd IEEE/ACM International Workshop on Software for Sensor Networks (SensorWare 2007), 2007.
- [22] Sameer Tilak, Nael B. Abu-Ghazaleh, and Wendi Heinzelman, *A taxonomy of wireless micro-sensor network models*, SIGMOBILE Mob. Comput. Commun. Rev. **6** (2002), no. 2, 28–36.
- [23] Matt Welsh and Geoff Mainland, *Programming sensor networks using abstract regions*, Proc of 1st Symposium on Networked Systems Design and Implementation, 2004.
- [24] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler, *Hood: A neighborhood abstraction for sensor networks*, Proc of 2nd International Conference on Mobile Systems, Applications, and Services, 2004.
- [25] A. Woo, S. Seth, T. Olson, J. Liu, and F. Zhao, *A spreadsheet approach to programming and managing sensor networks*, Proc. of the International conference on Information processing in sensor networks, 2006.