

國立臺灣大學電機資訊學院資訊工程學研究所

碩士論文

Graduate Institute of Computer Science and Information Engineering

College of Electrical Engineering & Computer Science

National Taiwan University

Master Thesis

基於模型的可延展性組態工具

An Extensible Model-based Configuration Tool



陳俊衛
Chen, Chun-Wei

指導教授：薛智文 博士

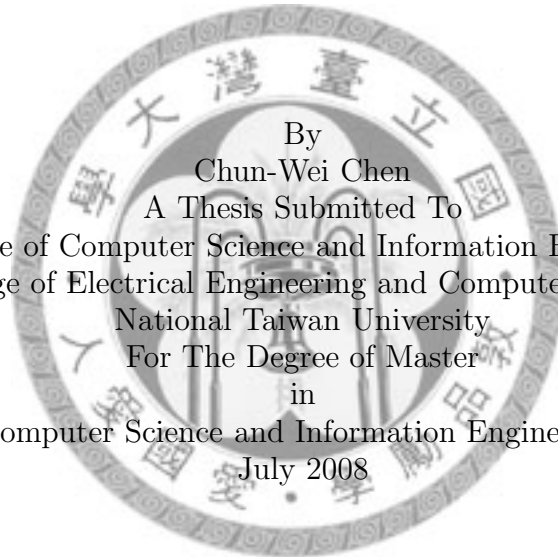
Advisor : Hsueh Chih-Wen, Ph.D.

中華民國 97 年 7 月

July, 2008

An Extensible Model-based Configuration Tool

By
Chun-Wei Chen
A Thesis Submitted To
Institute of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
For The Degree of Master
in
Computer Science and Information Engineering
July 2008



基於模型的可延展性組態工具

指導教授：薛智文 博士 研究生：陳俊衛

國立台灣大學資訊工程學研究所



摘要

在許多不同的領域中都需要組態，像是處理器的組態，產品組態，軟體組態等，這些不同的組態都有其特定的專業知識。因此，延展性與彈性成為了組態工具的重大需求。然而，現今並沒有組態工具是設計來同時對應多個不同的領域。在這篇碩士論文中，我們提出了基於模型的可延整性組態工具 (EMC Tool)，可以簡單地整合新的領域。我們的工具採用了 EMCXML 作為輸入的規格，可以將專業知識用統一的 XML 元素模型化；同時我們也採用了模組化的設計，讓使用者在不同的狀況下能有更高的彈性。我們相信我們工具的可延展性與彈性可以減少開發者冗長的組態工作。

關鍵字：組態，領域，專業知識，XML，模組化

Abstract

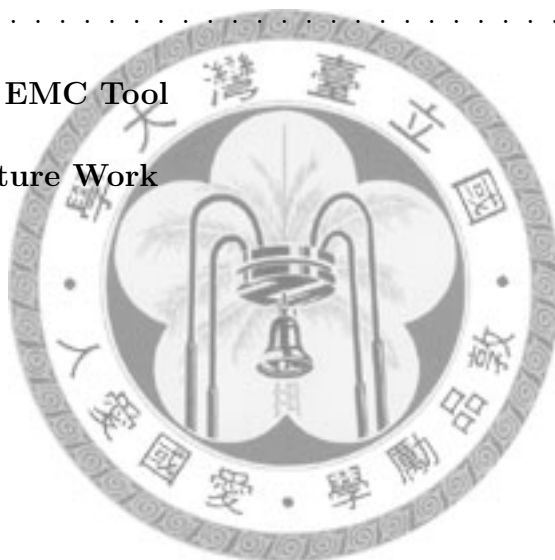
Configuration is necessary in various domains, such as processors configuration, product configuration, and software configuration where the specific domain knowledge is required. Therefore, being extensible and flexible becomes a heavy demand for configuration tools. However, no configuration tool is designed for various domains simultaneously nowadays. In this thesis, we propose an Extensible Model-based Configuration Tool (EMC Tool) to allow easy integration of new domains. Our tool uses EMCXML as input format where the domain knowledge is modeled as uniform XML elements such that configuration can be easily created and extended. Also we adopt modularized design to achieve flexible configuration for users under different situations. We believe that such extensibility and flexibility of our EMC Tool will reduce the complexity of tedious configuration work for future developers.

Keyword: configuration, domain, domain knowledge, XML, modularize.

Contents

1	Introduction	1
2	Background	6
2.1	Configuration in Different Reasoning	6
2.1.1	Rule-based Reasoning Configuration	7
2.1.2	Model-based Reasoning Configuration	8
2.1.3	Case-based Reasoning Configuration	9
2.2	Generic Models of Configuration Tasks	10
2.2.1	General Definition of Configuration Tasks	11
2.2.2	Restricted Version of Configuration Tasks	12
2.3	Domain Knowledge Modeling	13
2.3.1	Components	13
2.3.2	Functional Architecture	14
2.3.3	Mapping from Functions to Components	14
2.4	Related Work	15
2.4.1	VEST	15
2.4.2	AADL	16
3	Architecture	18
3.1	Goals of the EMC Tool	18
3.2	EMCXML	19
3.2.1	Key Components	21
3.2.2	Display on Configuration Tool	23
3.2.3	Data Type	23
3.2.4	Output Information	25

3.2.5	Ports and Children	26
3.2.6	Constraint	27
3.2.7	EMCXML Schema	28
3.3	Extensible Model-based Configuration Tool	30
4	Implementation	35
4.1	Input Layer	35
4.2	Configuration Description Layer	36
4.2.1	Apache Xerces2 Java Parser	38
4.2.2	Google Web Toolkit	39
4.2.3	Implementation of Element <i>dependency</i>	41
4.3	Output Layer	42
5	An Example using EMC Tool	44
6	Conclusion and Future Work	48
	Bibliography	49



List of Figures

1.1	Two key features of configuration	2
2.1	Example of Models in Embedded Systems Design	9
3.1	Flow Chart of The EMC Tool	19
3.2	EMCXML Format	22
3.3	A Simple Example of The Element <i>dependency</i>	25
3.4	A Complex Example of The Element <i>dependency</i>	26
3.5	The relationship between ancestors and descendants is a tree graph	27
3.6	An Example of The Element <i>Relationship</i>	28
3.7	An Example of EMCXML Schema	29
3.8	Architecture of Extensible Model-based Configuration Tool	31
3.9	An Example of Output Layer	33
4.1	XAmple XML Editor	37
4.2	Output Information from Configuration Description Layer to Output Layer	43
5.1	Linux Booting Flow	45
5.2	An EMC XML Example of Describing BSP	46
5.3	Configuring BSP Using EMC Tool	47

Chapter 1

Introduction

Configuration is a special type of design activity [11] with two key features:

- The artifact being designed is assembled from instances of a set of well-defined components,
- Components can only be connected together in pre-defined ways.

This intuitive definition fits a large number of design tasks from our daily life. Not only in the high technology areas, many tasks are also configurations, such as planning a set of actions for achieving specific goals, developing a therapy as a composition of cure and synthesizing problem-solving strategies. Moreover, product configuration in different domains [18], configuration management, configurable operating system and integration of hardware and software in embedded systems are configuration tasks. Therefore, it is no surprise that configuration has become much more significant.

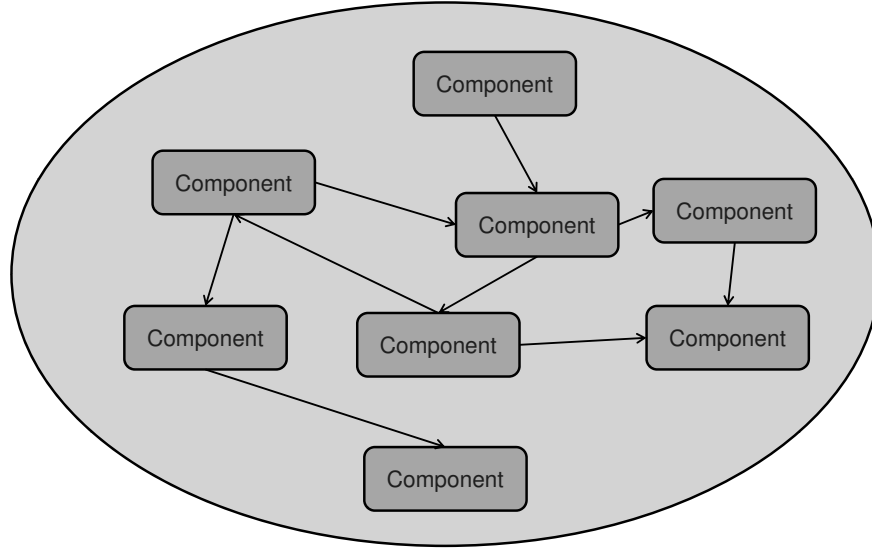


Figure 1.1: Two key features of configuration

After the abstract meaning of configuration is defined, we focus on the concretion of configuration: configuration tasks. The specification of a configuration task involves at least two distinct phases: describing the domain knowledge and specifying the desired target [15]. The domain knowledge describes the object types available in the application domain and the relations within object instances. After the first phase is finished, the environment of the desired target is sketched. Then requirements of the desired target will be proposed and must be satisfied. A special characteristic of configuration tasks is that the problem has already been well-defined. In other words, all components are defined completely, and the relations between distinct components

are described explicitly. However, to represent a configuration task is not quite simple. Most of the complexity of solving a configuration problem lies in representing domain knowledge; therefore, how to represent the domain knowledge is extremely critical.

Nowadays, many applications are designed to model a range of domain knowledge and are used to configure specific targets. This kind of applications contains built-in domain knowledge and usually has a particular aim. For instances, Kbuild [8] is used to manage configuration in Linux kernel, Apache configuration tool [19] is used to help system administrators to tune the main Apache configuration file, etc. Nevertheless, when we face a much wider condition involving large range domain knowledge, these applications are not sufficient for our requirements due to the domain knowledge is built-in and hard to modify or extend. This kind of applications lacks flexibility and extensibility to face even a little more complex situation.

To avoid this problem, an easy approach is to separate domain knowledge from applications. Many languages and models such as CLASSIC [2], CML2 [3] and etc, were proposed for describing domain knowledge of configuration tasks. However, they are usually bounded in their domain, i.e., if we want to extend our domain into a wider scenario, this kind of models will not be sufficient. Moreover, to learn new native languages for configuration takes a great deal of time, and the learning efforts are heavy. Besides, the native

language structure lacks expressiveness due to they are not in a common public format, this characteristic will easily make developers confused.

In order to solve the problem described above, we propose an extensible model-based configuration tool (EMC Tool) which emphasizes on both phases of configuration, i.e., we not only propose EMCXML schema for describing domain knowledge with more expressiveness and shorter learn curve, but also provides a configuration GUI for users accordingly. With the public XML documents, the developers can easily understand how to use our models instead of taking a lot of efforts, and resources about XML is very easy to find. They do not have to learn new languages and only need to understand XML models. After developers define their own domain knowledge by following EMCXML schema, The EMC Tool can build configuration environments. Finally, developers can easily obtain their configuration results from our EMC Tool, and do the following activities. According to the modularized design, the EMC Tool has more flexibility and extensibility. For example, the EMC Tool is able to output to different types of files, i.e, video, audio, text file, etc, by altering different output modules. Developers can choose the most appropriate modules for their sake. By separating domain knowledge from itself and the modularized structure, our EMC Tool has more wide ability. Our configuration tool provides an easy way for configuration tasks from describing domain knowledge to generating ultimate results. We believe that the complexity of configuration can be reduced by using our tool.

The rest of the thesis is organized as follows. The next chapter introduces the background information about different types of configuration and a general model approach. Section 3 details how to describe the domain knowledge by EMCXML schema and also introduces the architecture of our configuration tool. The implementation is described in Section 4. Consequently, conclusion and future work are given in Section 6.



Chapter 2

Background

In this chapter, we provide the background information before we introduce the EMC Tool. Firstly, we introduce different types of configuration for different reasoning in Section 2.1. Secondly, we discuss about the generic model for the configuration tasks in Section 2.2. Finally, we point out related work of our research.

2.1 Configuration in Different Reasoning

An abstraction configuration depends on how it represents. The most common types of configurations are rule-based reasoning configuration, model-based configuration, and case-based reasoning. The following sections introduce these three kinds of configurations.

2.1.1 Rule-based Reasoning Configuration

Expert systems use production rules as a uniform mechanism for representing not only domain knowledge, but also control strategies. In a rule-based reasoning system, production-rule programming languages are usually adopted. This kind of languages involves "if-then-else" rule statements which are simple patterns, and can be searched in an inference engine by matching patterns. The "if" indicates a condition. If the condition is true, the "then" action will be executed; if the condition is false, the "else" action will be executed instead of the "then" element.

There are two common organizations for rules: one is forward-chaining, also known as data-driven, and the other is backward-chaining, also known as goal-directed [6]. In forward-chaining, at each step, the system starts with the rules and considers only rules that can be executed. It repeats this action until the goal is reached. Contrarily, backward-chaining starts from a goal, and looks for rules which can apply to the goal until a conclusion is reached.

Unfortunately, rule-based reasoning bounds in a limited understanding of configuration process. The existing rule-based systems were designed to solve specific instances of configuration tasks [15] because no common rules can be applied in general situations.

2.1.2 Model-based Reasoning Configuration

Model-based reasoning is based on a knowledge base that describes a particular problem area in terms of the behavior of its small building components. The main assumption behind model-based reasoning is the existence of a system's model, which consists of decomposable entities and pre-defined interactions between their elements. In other words, the model is the essential element in a model-based system during the configuration time. The most important advantages of model-based systems are

1. A better separation between what is known and how the knowledge is used
2. Increasing ability to solve a broader range of problems
3. Increasing ability to combine knowledge from different domains within a single model
4. Increasing ability to use existing knowledge to solve related classes or problems

The major motivation for using model-based configuration system is that configuration in the nature is by definition a synthesis task. Therefore, to cover the entire range of solutions is an essential ability. Although a system can gain experience in a particular domain and use it to improve efficiency, the system should still work without prior experience. Many systems are published to configure a particular domain, i.e. networks, operating systems, vehicles,

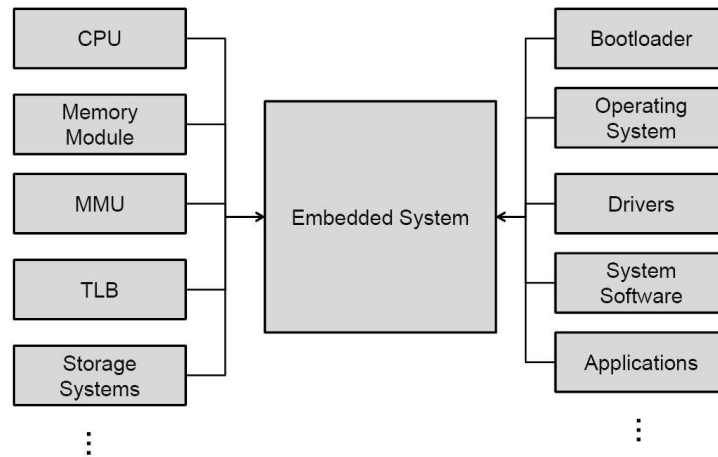


Figure 2.1: Example of Models in Embedded Systems Design

circuit boards and etc. By modeling the domain knowledge, a complex system can be divided into several simple and flexible parts, so that developers will easily understand the behavior of the complex system without much effort. Figure 2.1 shows an example of models in embedded systems design which involves hardware and software.

2.1.3 Case-based Reasoning Configuration

Case-based reasoning is very different from other reasoning technologies presented above. The process of solving a configuration problem by case-based reasoning relies on the solutions of previously similar problems. With case-based reasoning, the current configuration problem attempts to solve by finding a set of solutions in similar past problems and adjust them to the current configuration. By observing the methodology of case-based reasoning, we can find that case-based reasoning is heavily based on a assumption: Similar problems

have similar solutions.

As the highest level of generality, a general case-based reasoning cycle can be described by the following four steps [1]:

1. Retrieve the most similar case or cases
2. Reuse the information and knowledge in the case to solve the problem
3. Revise the proposed solution
4. Retain the parts of this experience likely to be useful for future problem solving

2.2 Generic Models of Configuration Tasks

In section 2.2, we introduced the model-based reasoning configuration. A central issue in knowledge presentation is to effectively describe a complex scenario, so that ones can use the description for flexible problem solving. Therefore, in a model-based system, the most critical thing is how to model the domain knowledge. If a general modeling of configuration tasks is existed, the complexity of solving configuration problems will be reduced. However, it is not easy to build a general model due to the large difference of distinct domains. In the following sections, we will discuss how to build a general model for configuration tasks.

2.2.1 General Definition of Configuration Tasks

Making very few assumptions about the kinds of knowledge that might be available; we define a configuration task the same as [11] as follows:

Given three premises:

1. A fixed, pre-defined set of components, where a component is described by a set of properties, ports for connecting it to other components, constraints at each port that describe the components that can be connected at that port, and other structural constraints.
2. Some description of the desired configuration.
3. Possibly some criteria for making optimal selections.

Build:

- One or more configurations that satisfy all the requirements, where a configuration is a set of components and a description of the connections between the components the set, or, detect inconsistencies in the requirements.

There are three significant aspects to this definition. The first is, the components are fixed, i.e., no new component can be designed. The second is, the interactions between different components are in fixed and pre-defined way. In other words, the components cannot be modified their connectivity. "Port" indicates an abstraction places where a component can be connected to other

components. Consequently, the third is that a solution not only specifies the actual components, but also shows how to connect them together.

The definition above clearly provides a standard of modeling for configuration tasks. For examples, a customer buying a car is actually configuring from a set of components: car models, engines, brakes, etc. According to the customer's imaginary description for his desired car (e.g. 250 horsepower, 3000 c.c. displacement engine, red car module looks, etc), he/she would select his/her favorite choices, then the solution will occurs. Similar examples can be found in other domains.

2.2.2 Restricted Version of Configuration Tasks

We now introduce two restrictions on the general configuration task. These restrictions reduce the complexity of the task and help in identifying additional kinds of knowledge.

The first restriction is that the artifacts are configured according to some known functional architectures. In other words, instead of trying to assemble all possible artifacts that can be created from the given set of components, one restricts the problem to those artifacts that are similar in their architecture. This clearly restricts the scope of the task but not in an arbitrary way.

The second restriction is the key components per function. A component can identify some particular component that is crucial to implement some func-

tions. For example, the printing function needs a printer component. Other components needed for the printing function such as hardware interface, data cables, power cable, fonts, and driver software can be determined once a printer has been selected. Thus, we do not need to consider each configuration for printing functions, we only need to start with a printer and build suitable configurations from there. In this case, the printer component is the key component of printing function.

2.3 Domain Knowledge Modeling

2.3.1 Components

Components can be described independent of how they are used by a set of physical properties, ports and constraints. Physical properties, for example, in an LCD, includes resolution, viewable size and so on. Ports indicate an abstraction place where other components can attach, e.g., typical ports for an LCD include data port, power supply port, image decoder port etc. Ports themselves can be described by some set of properties. Constraints have the ability to limit what components can be attached there. Typically, these constraints would describe properties of the components that can be connected at that port or more specifically properties of a port on another component. Thus, an LCD with a "220V voltage power supply port" would have a constraint that any component attaches to this port must match this voltage constraint. Another ability of

constraints is to give the limitation of some physical properties. As the voltage power supply port mentioned above, it's constraint is it only works when the power is 110V or 220V.

Components are the essential elements in domain knowledge describing. By definition [11], Components cannot be modified and created during configuration time. The relationship between distinct configurations cannot be modified either. All components are defined before configuration time and will be picked or assembled during the period.

2.3.2 Functional Architecture

A functional architecture specifies a functional decomposition of the artifacts and constraints on their composition. Individual functions can be simply modeled by a set of properties that characterize them. For instance, the display function may be described by properties such as source, contrast, resolution and illumination.

2.3.3 Mapping from Functions to Components

Finally, we need to model the knowledge for mapping from functions to components. A function can be implemented by a set of components. For example, the display function needs components such as an LCD, a cable line, and a signal source. On the other hand, components are often multi-functional.

The key component per function can help us to simplify the representation. Functions can be indexed via their key component, so those functions will not be ambiguous and confounded. In the former example, the key component of the display function is its LCD component. So by select the LCD component, we also select its corresponding unique function. Other components in the function will be selected automatically after the LCD component is selected immediately.

2.4 Related Work

In this section, we introduce some famous related configuration tools. They are popular in their domain and really helpful. However, in spite of they are very success in their domain, but they do not satisfy our requirement: domain knowledge free. Nevertheless, they still have a great amount of reference value for our design.



2.4.1 VEST

VEST [17] provides an environment for the composition and analysis of distributed real-time embedded systems. VEST models application components, middleware, OS, and hardware components. This feature supports the composition and tailoring of every layer in an embedded system for a specific application, which leads to more complete crosscutting dependency checks and more optimization opportunities. VEST itself is not a complete requirements, design

and implementation tool; rather it currently focuses on the specific composition and analysis tasks.

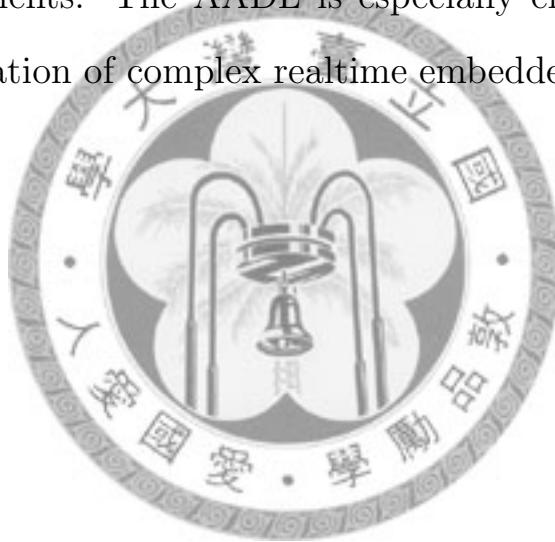
VEST includes features that are found in other tools. However, there are several novel features in VEST. The major contributions of VEST are two types of language-independent aspects referred to as aspect checks and prescriptive aspects. Together these permit the benefits of aspects to be exercised early in the composition process rather than in the implementation phase. A set of representative aspect checks in embedded software is identified and implemented in VEST. Some of these aspects are simple dependency checks; others are complex and may involve the entire system, e.g., distributed real-time scheduling. The simple fact of identifying key aspect checks improves our understanding of specific crosscutting concerns found in distributed embedded systems, including middleware. Prescriptive aspects allow application specific advice to be applied to designs and they have a global effect. The significance of VEST is largely derived from language-independent aspects.

2.4.2 AADL

In November 2004, the Society of Automotive Engineers (SAE) released the aerospace standard AS5506, named the Architecture Analysis and Design Language (AADL) [5]. The AADL is a modeling language that supports early and repeated analyses of a systems architecture with respect to performance-critical properties through an extendable notation, a tool framework, and precisely de-

finer semantics.

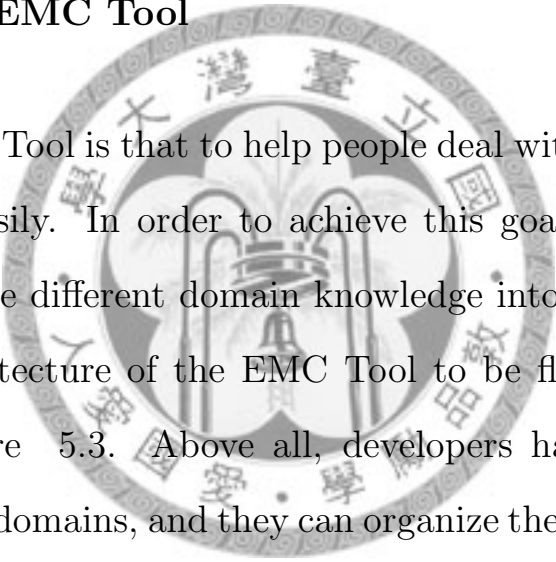
The language employs formal modeling concepts for the description and analysis of application system architectures in terms of distinct components and their interactions. It includes abstractions of software, computational hardware, and system components for (a) specifying and analyzing real-time embedded and high dependability systems, complex systems of systems, and specialized performance capability systems and (b) mapping of software onto computational hardware elements. The AADL is especially effective for model-based analysis and specification of complex real-time embedded systems.



Chapter 3

Architecture

3.1 Goals of the EMC Tool



The goal of the EMC Tool is that to help people deal with various configuration tasks rapidly and easily. In order to achieve this goal, we not only propose EMCXML to describe different domain knowledge into a uniform format, but also design the architecture of the EMC Tool to be flexible. The flow chart is illustrated in figure 5.3. Above all, developers have some configuration problems in different domains, and they can organize their configuration domain by EMCXML at this time. The EMC Tool admits EMCXML to be its input format and contains no domain knowledge, therefore the EMC Tool can adopt to a great deal of domains via EMCXML. Sequentially, a flexible architecture to adapt to different situations is necessary. The EMC Tool is designed as a modularized structure for this reason.

Our work can be divided into two parts: EMCXML and the EMC Tool. Both of them are extensible and will be discussed in Section 3.2 and Section

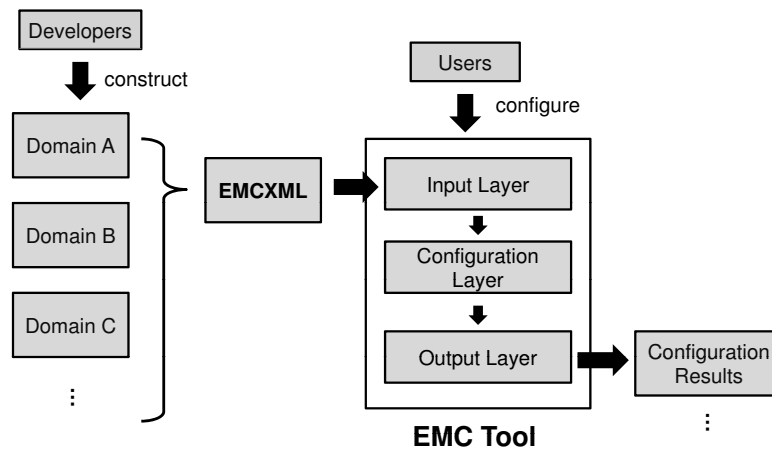
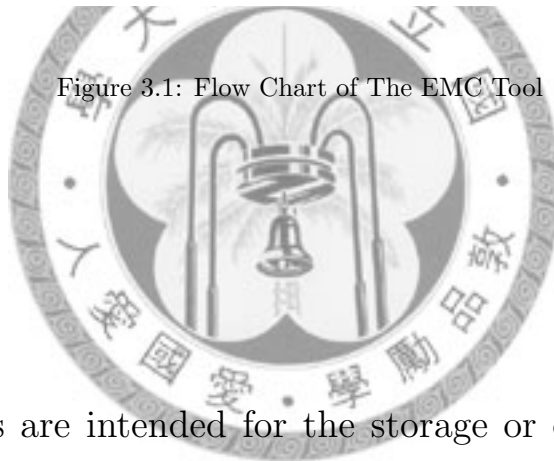


Figure 3.1: Flow Chart of The EMC Tool

3.3.

3.2 EMCXML



XML [7] documents are intended for the storage or exchange of data or information, and can be used to store data that would traditionally be stored as documents, letters, reports, manuals and so on or data that might associate with databases. A part of strength in XML is, at least for developers, human-readable. Typically, an XML developer uses element type names (or called tag names) that are meaningful. For all these reasons, the World Wide Web Consortium (W3C), the de facto Internet standards body, undertook to create XML starting in 1996. In order to design to overcome limitations in HTML,

XML is based on an ISO standard, Standard Generalized Markup Language (SGML). The XML 1.0 Recommendation was approved in 1998 end and, like many XML-related recommendations, is currently maintained by the W3C.

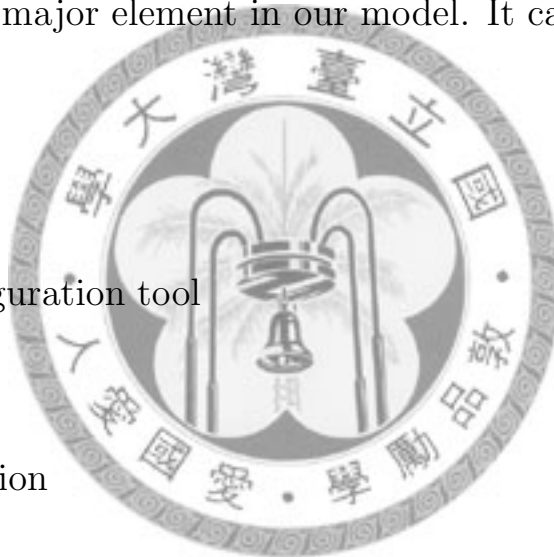
There are many choices to be our configuration description languages, such as regular expressions or native languages. We have many language candidates, but why we choose XML to be our format? The answer consists of four points. The first point is, XML is a general-purpose specific language, in the other hand, we can define our own XML format for our sake. The second point is, XML is a markup language, which provides well representation power, therefore the complex scenario of configuration could easily be described by XML. The third point is, XML is a common public standard, many resources (e.g. parsers, editors, etc) exist and easily be found. This characteristic could efficiently reduce the effort of developing XML-based applications. The fourth and final point is, XML is full of human-readability. This advantage makes developers use XML to construct their domain knowledge much easier. If they use other representations such as regular expressions or native languages, they have to pay a lot of time to learn. XML has a shorter learning curve, and all these four characteristics will attracts many users to choose our approach. That is why we select XML to be our format.

We define EMCXML (Extensible Model-based Configuration XML) to describe specification of configurations and to be the format of the configuration

description file is shown in figure 3.2. All the configurations are under the element *main* and can be divided into two categories: the configuration definitions and the overall output information, whose corresponding elements are *config* and *outputfile*. These two elements indicate two functions (the definition of functions was defined in Section 2.3.2), and all the elements under them are components.

The element *config* indicates configuration, which includes a set of properties, ports, are the major element in our model. It can be divided into seven categories:

- Key components
- Display on configuration tool
- Data type
- Output information
- Ports
- Children
- Constraints



3.2.1 Key Components

As mentioned in Section 2.2.2 and Section 2.3.3, key components are indispensable in a function. We define each configuration as a function; therefore,

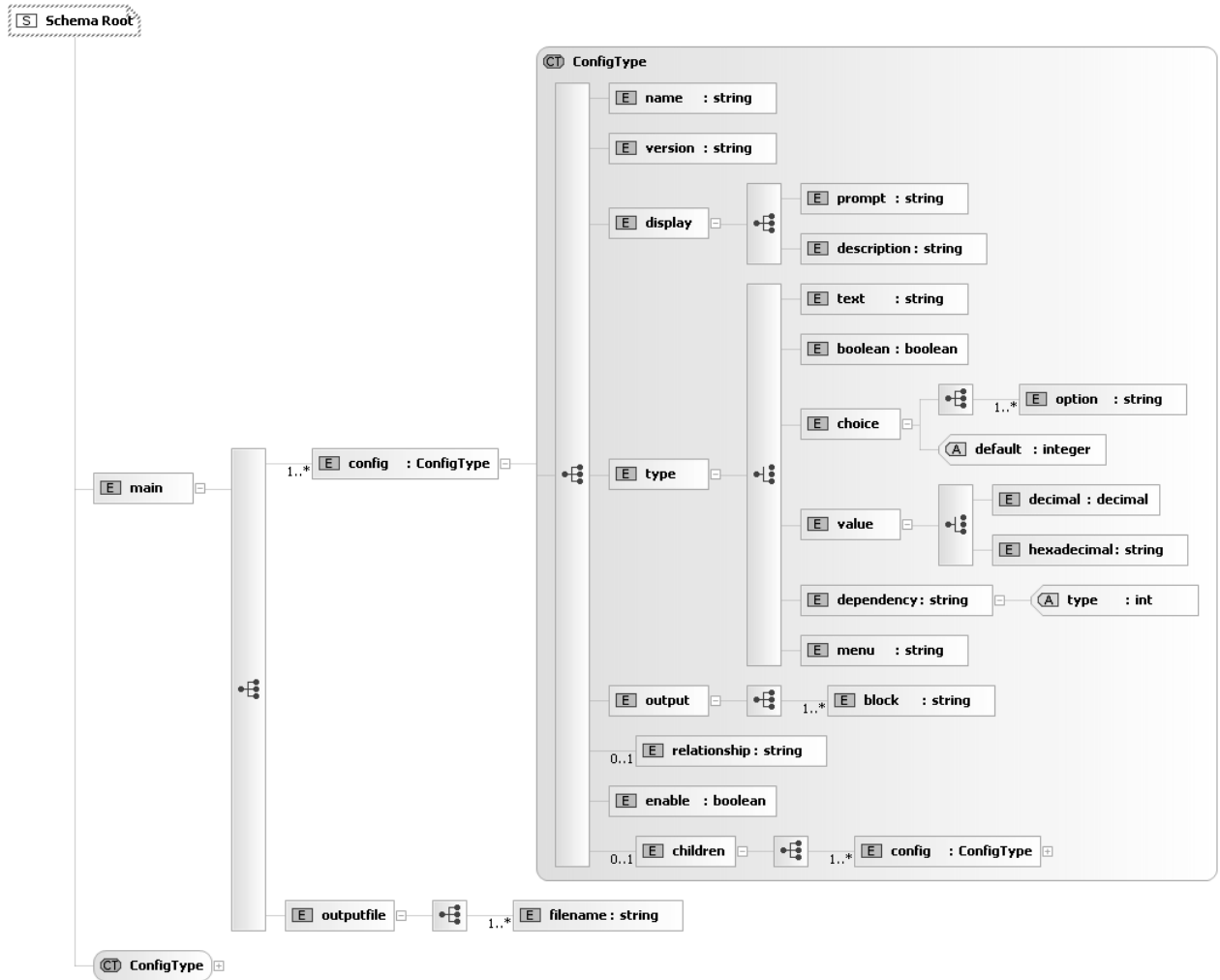


Figure 3.2: EMCXML Format

the element *config* needs key components to indicate its index. For this purpose, the element *name* is defined to be the key component. Each element *name* is unique in the whole configuration description file, hence we can easily distinguish configuration. However, we may modify configuration into a newer version. In this case, the elements *names* in both old and new configurations are identical. To solve this problem, another element *version*, which indicates the current version of its configuration, is added into the EMCXML. Conclusively, the key components in configuration are the elements *name* and *version*.

3.2.2 Display on Configuration Tool

To display each configuration on the EMC Tool needs related information. The element *display* and its children elements *prompt* and *description* focus on this task. The element *prompt* is given for each configuration, and it can point out what the configuration does briefly. Subsequently, the title of each configuration will be shown on our EMC Tool in terms of its element *prompt*. In addition, users may need detailed information about configuration. For the reason, the element *description* is proposed to store the detail of configuration.

3.2.3 Data Type

The most significant part of a configuration is its data type. The element *type*, which defines data types in EMCXML, includes six categories of children elements to indicate different types:

- *text*
- *boolean*
- *choice*
- *value*
- *dependency*
- *menu*

The element *text* means the data of a configuration is a text string, and *boolean* means that the data of a configuration is a Boolean value: either true or false. *Choice* indicates that the data of a configuration consists of multiple options, and then users can select a proper one. *Value* describes a the data type of configuration is a numeral, and it contains two children elements *decimal* and *hexadecimal* to represent the radix. The element *menu* presents a configuration is a menu object in the configuration description file when the configuration contains no data but children configurations.

The element *dependency*, which is very different from the other five data types, defines that the configuration's data depends on other configurations. In order to achieve this goal, we can describe a configuration's value as a series of script. For example, from the simple start, configuration A can be multiplied by configuration B and configuration C, then configuration A can be described as figure 3.3. Furthermore, instead of a simple expression, the element *dependency*

```

<name> A </name>
...
<type>
    <dependency>
        ~B~ * ~C~
    </dependency>
</type>
...

```

Figure 3.3: A Simple Example of The Element *dependency*

provides a more powerful functionality by script language as shown in figure 3.4. Therefore, by the element *dependency*, the value of a configuration can be set more flexibly.

3.2.4 Output Information

In order to store output information or activities, the element *output* is defined for this purpose. A configuration can not only output to a specific target, but also multiple ones. The output information corresponding to each output target is stored in the element *block* which is the child element of *output*. In the other hand, a *block* indicates output information or activities to a specific output target.


```

<name> A </name>
...
<type>
  <dependency>
    if ~B~ > 10
      ~A~ = ~C~ * 5
    else
      ~A~ = ~C~ * 2
  </dependency>
</type>
...

```

Figure 3.4: A Complex Example of The Element *dependency*

3.2.5 Ports and Children

The meaning of ports of a configuration was defined in Section 2.3. In EMC Tool, there are two kinds of relations: implicit tree structure and explicit relations defined by users.

A configuration can naturally own its children and descendant. Via the element *children*, we can define the children configurations of each configuration. The ability of descendant configurations inherits their ancestor configuration, and hence descendants are not available until ancestors are available. The relations of ancestors and descendants become a tree graph intuitively and implicitly as shown in figure 3.5.

The elements *relationship* and *enable* are used to present connections and relations between configurations. *Enable* indicates that the configuration is

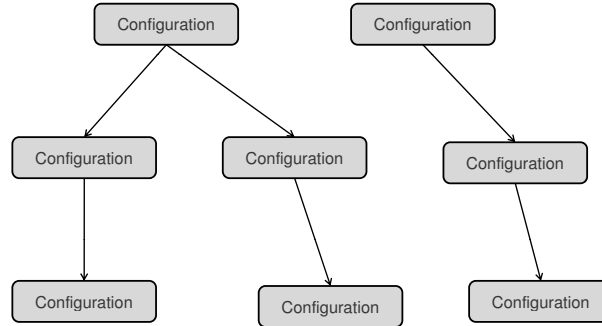


Figure 3.5: The relationship between ancestors and descendants is a tree graph

either available or unavailable. The element *relationship* describes the explicit relations between different configurations. Via these elements, a configuration is able to influence other configurations's availability even they are not paternity. For example, in figure 3.6, first we notice the symbol $\sim B$, which indicates configuration B and will return the value of B's *enable* element. Then we can find that the configuration A is always unavailable in terms of its *relationship* element always returns false (true AND false equals false). The return value of the element *relationship* will overwrite the value of *enable*.

3.2.6 Constraint

In order to limit the value or functionality of configuration, we can give configuration the element *constraint*. By following the element *constraint*, the complexity of configuration can be reduced. For example, we can define a con-

```

<name> A </name>
...
<relationship>
    ~B~ && !~B~
</relationship>
...

```

Figure 3.6: An Example of The Element *Relationship*

figuration as an integer. However, if we don't limit its value, then the variety of the configuration is infinite. In order to avoid the situation, we can give the configuration a constraint which is the integer must be equal or less than three and could not be negative. Therefore, the value of the configuration can only be zero, one, two or three, and hence the complexity is reduced greatly. Another advantage of the element *constraint* is, by setting a constraint to each configuration, users will take fewer mistakes cause of typo or confused. The element *constraint* can reduce the complexity of configuration.

3.2.7 EMCXML Schema

XML Schema expresses shared vocabularies and allow machines to carry out rules made by people. They provide an approach for defining the structure, content and semantics of XML documents in more detail. XML Schema was approved as a W3C Recommendation on 2 May 2001 and a second edition incorporating many errata was published on 28 October 2004 [16]. An XML

```

<?xml version="1.0" encoding="utf-8"?>
<xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="main">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="1" maxOccurs="unbounded"
          name="config" type="ConfigType" />
        <xs:element name="outputfile">
          <xs:complexType>
            <xs:sequence>
              <xs:element minOccurs="1"
                maxOccurs="unbounded" name="filename">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:minLength value="1" />
                  </xs:restriction>
                </xs:simpleType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="PK_config">
      <xs:selector xpath="./config" />
      <xs:field xpath="name" />
    </xs:key>
    <xs:key name="PK_menu">
      <xs:selector xpath="./menu" />
      <xs:field xpath="name" />
    </xs:key>
    ...
  </xs:element>
</xmlns:xs>

```

schema is also in XML document format, hence XML schema has the same features as XML; therefore, XML schema is easy to modify.

EMCXML may not sufficient to all conditions, in order to adapt to more situations, EMCXML could be extended. The extensibility is from the ability of XML schema, which gives XML document flexibility and agility. Therefore, EMCXML is not a fixed standard, developers can add new features into and modify by their own. EMCXML is extensible conclusively. A part of EMCXML Schema is shown in figure 3.7

3.3 Extensible Model-based Configuration Tool

As shown in figure 3.8, the extensible model-based configuration tool is a modularized structure, which consists of three major layers: input layer, configuration description layer, and output layer. Developers who build domain knowledge, will input information by EMCXML in input layer to configuration description layer. Configuration description layer receives EMCXML and creates a corresponding GUI, and users can configure via the GUI. After finishing configuration, the output layer will generate output by the users' choices. Developers and users are different groups of people. Developers are who build domain knowledge and will construct their knowledge by EMCXML. Different from developers, users only want to do configuration, they do not care about EMCXML and only care about final results.

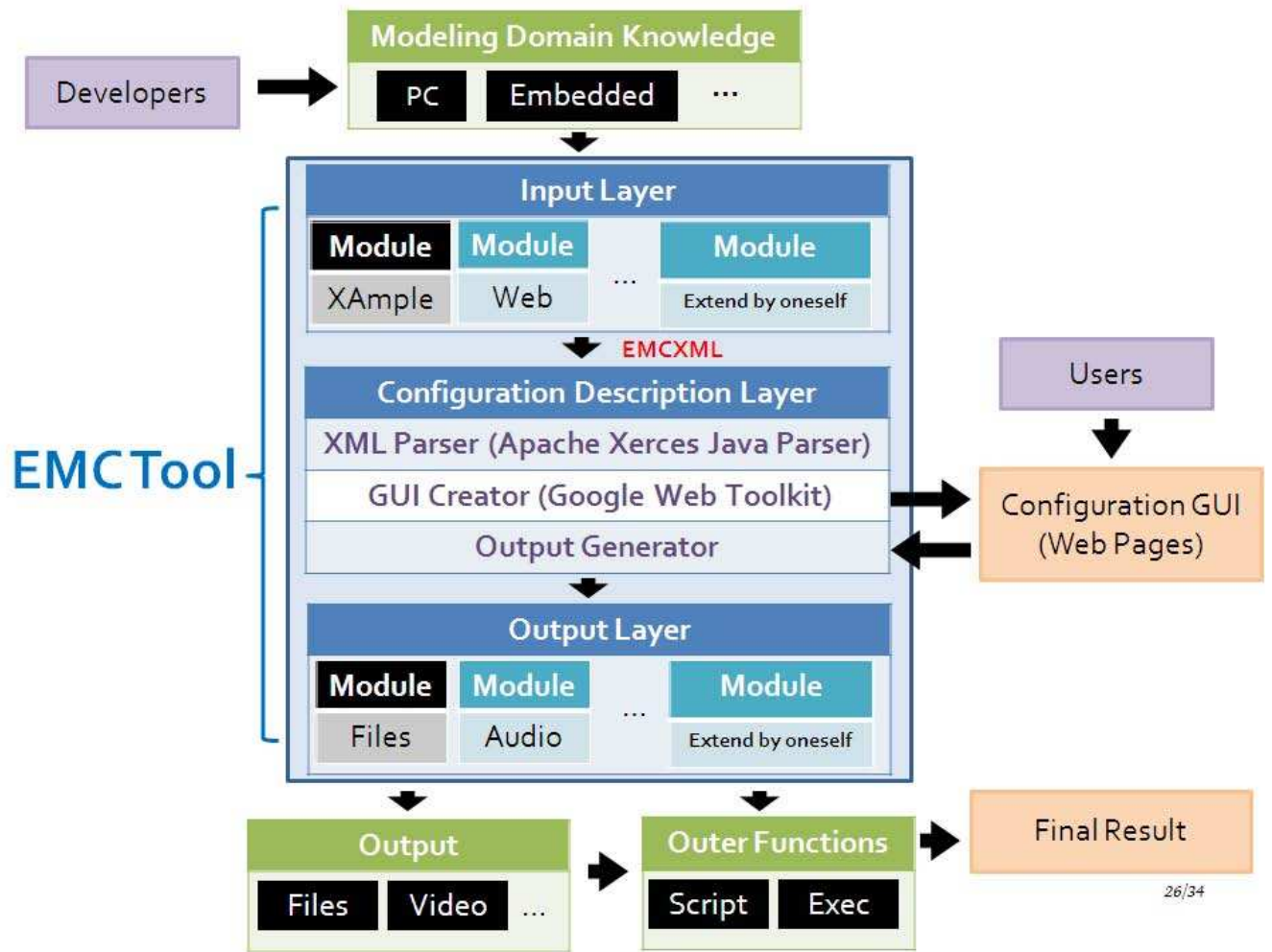


Figure 3.8: Architecture of Extensible Model-based Configuration Tool

The main goal of the input layer is to input XML information to the configuration description layer. Due to our domain knowledge is described by EMCXML, our tool provides a layer which focuses on XML input by following EMCXML format. As shown in figure 3.8, the input layer allows different modules inside. For instances, the input method may be by a web browser or speech input. All the different modules can be designed for different situations, but they have to follow EMCXML schema. Although modules are different, but their goal are consistent: build domain knowledge by EMCXML.

The configuration description layer is the core layer of the EMC Tool. This layer consists of three components: XML parser, user interface creator and output generator. The domain knowledge is modeled by XML; however, the XML document cannot be processed directly, therefore XML parser is the first component in the configuration description layer and is used to parse XML data into useful information. The core engine is inside of the XML parser. No matter data dependency, configuration relationship, or anything else, this kind of complex scenario is also handled in the XML parser. In the other hand, XML parser is the core data processing engine in the configuration description layer. After the EMC Tool parses the XML document and obtains the domain knowledge, it will create a corresponding configuration GUI. Output generator in the configuration description layer receives feedback from users via configuration GUI, and then passes the information to output layer.

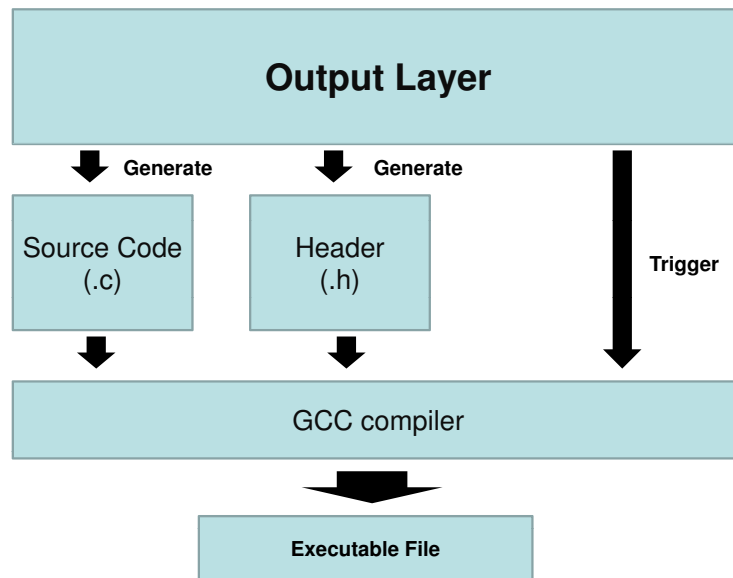


Figure 3.9: An Example of Output Layer

The output layer focuses on output information to specific targets which can be files or input of an application. The output layer can also trigger outer functions after users finishing their configuration. This layer is also a module and full of flexibility. The ability of the output layer is not only output to multiple targets, but also trigger outer functions, i.e., a script, an executable file, etc. Due to the modularized structure design, the output layer can be many different. Hence the output ability of the EMC Tool is more widely, in other words, the EMC Tool can output to not only text files, but also video, audio, etc. Moreover, the EMC Tool is able to extend its ability by invoking an outer function. By the means, users can define the automatical procedure for the time after output, and hence they will save amount of time. An example

of output layer can be found in figure 3.9. Users want to configure their C language project currently. After they finish configuring, the output layer generates source code files and header files. Then output layer trigger an outer function: gcc compiler and compile the whole C project. Consequently, the final executable will be generated.

As described above, the extensibility of the EMC Tool stands on the modularized structure which is shown in figure 3.8. We can found that both input and output layer are full of flexible and can be easily extended. Developers can follow our format to build their own input or output modules. Therefore, we can declare our tool architecture design is extensible.



Chapter 4

Implementation

In this chapter, we explain the design concept of the EMC Tool and describe how we implement it. The framework of the EMC Tool has been presented in Chapter 3. For portable and easy-to-modify, the whole EMC Tool is implemented by Java, and the involved libraries also implemented by Java.

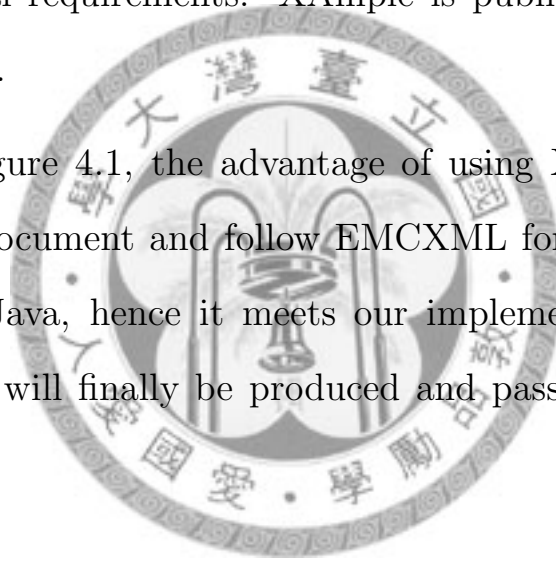
4.1 Input Layer

As we discussed in Section 3.3, input layer is used to model the domain knowledge. In the other hand, this layer focuses on input XML information to the configuration description layer. For this purpose, this layer needs to provide a user interface for developers, which not only has to be user friendly, but also follows the EMC XML format we defined in Section 3.2.

In order to achieve these two goals, we adopt an open freeware to be our input layer named XAmple XML Editor [4]. XAmple XML Editor Project introduces a java Swing based XML editor that analyzes a given schema and

then generates a document-specific graphical user interface. Unlike other XML editors, the XAmple XML editor GUI exposes not just a tree representation of the XML document but rather a logical combination of the XML document and respective XML Schema. The user interface of the XML editor is highly logical and intuitively comprehensible. To be able to prepare valid XML documents of significant complexity, a user is not required to be familiar with XML and XML Schema languages and to have any previous knowledge about the documents structural requirements. XAmple is published under the Apache Software License [9].

As shown in figure 4.1, the advantage of using XAmple is that we can easily edit a XML document and follow EMCXML format. Besides, XAmple is implemented by Java, hence it meets our implemental need. Ultimately, the XML document will finally be produced and passed to the configuration description layer.



4.2 Configuration Description Layer

Configuration description layer is the core layer of the EMC Tool. After developers model the domain knowledge, the configuration description layer will parse the XML document from input layer, and then create a configuration graphical user interface for users. Consequently the configuration description layer will receive the feedback from users and generate output information.

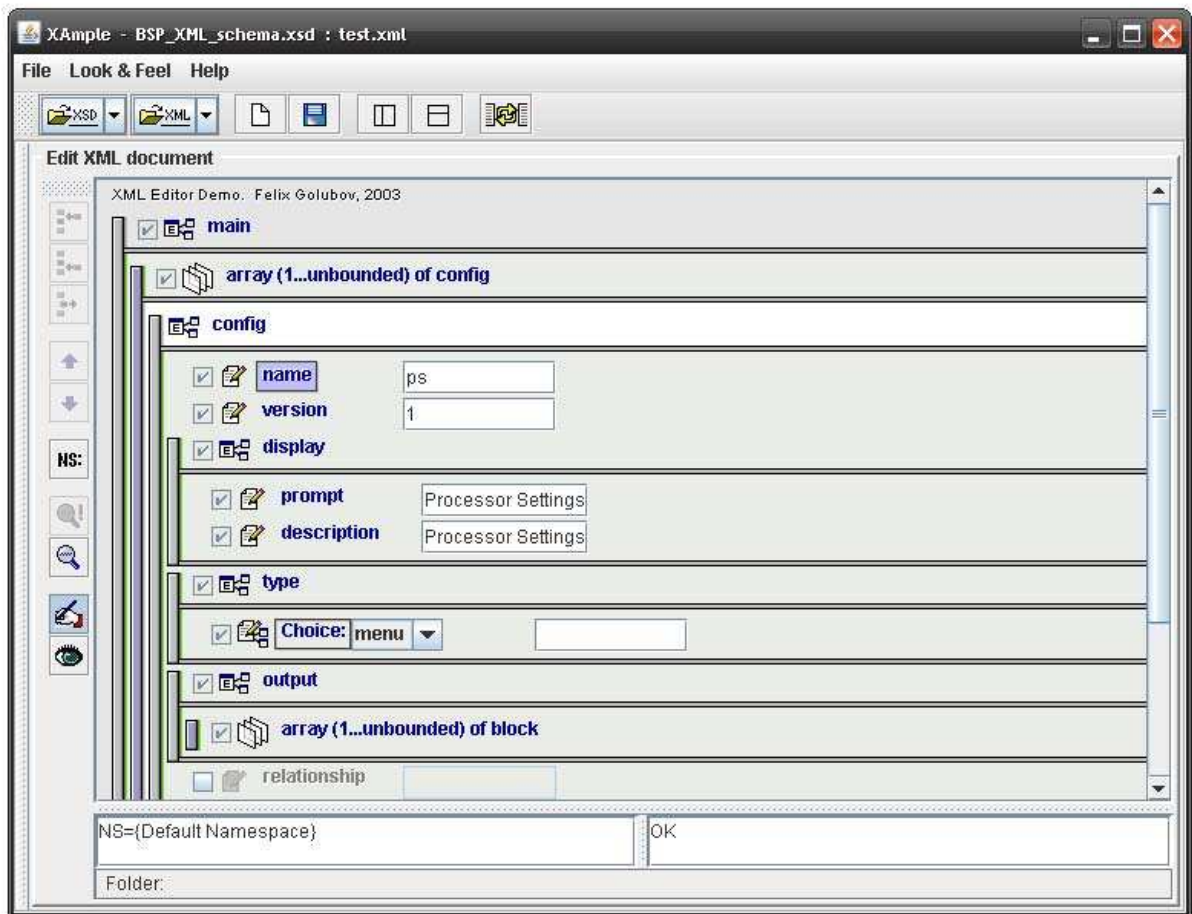


Figure 4.1: XAmple XML Editor

4.2.1 Apache Xerces2 Java Parser

The first component of configuration description layer is XML parser, which has to understand the EMCXML syntax and make a response according to EMCXML Schema.

Typically, there are two approaches to parse XML document: SAX and DOM. SAX which stands for Simple API for XML, is what makes insertion of this application-specific code into various events. The interfaces provided in the SAX package will become an important part of any programmer's toolkit for handling XML [10]. Even though the SAX library is small and few in number, they provide a critical framework for Java and XML to operate within. The main feature of SAX is that it is event-driven and parses the XML document once sequentially. Consequently, SAX is a light-weight library; hence its functions are easier. However, according to this feature, functions in SAX are also limited.

Unlike SAX, the Document Object Model (DOM) [12] which defined by W3C, is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing can be incorporated back into the presented page. Because the DOM supports navigation in any direction (e.g., parent and previous sibling) and allows for arbitrary modifications, an implementation must at least buffer

the document that has been read so far (or some parsed form of it). Hence the DOM is likely to be best suited for applications where the document must be accessed repeatedly or out of sequence order.

After the discussion between SAX and DOM, we tend to have a more functional toolkit to parse our XML document. When developers want to modify the XML parser module in the EMC Tool, flexibility and extensibility are much more significant. Because DOM provides a more flexible solution than SAX, for this reason, we adopt DOM to be our XML parser.

Apache Xerces2 Java Parser [13] is the next generation of high performance, fully compliant XML parsers in the Apache Xerces family. This new version of Xerces introduces the Xerces Native Interface (XNI), a complete framework for building parser components and configurations that is extremely modular and easy to program. In addition, Xerces2 supports both DOM and SAX, and we adopt its DOM library. Xerces2 is free software available under Apache Software License [9].

4.2.2 Google Web Toolkit

The second component in the configuration description layer is user interface creator. This component creates a configuration graphical user interface which provides a means for users to do their configuration according to the domain knowledge from the input layer. Therefore, we seek a toolkit that can build

GUI rapidly and dynamically.

Google Web Toolkit (GWT) is an open source Java software development framework that makes writing AJAX applications easy for developers who don't familiar with web-developing languages. Writing dynamic web applications today is a tedious and error-prone process; developers spend huge percent of their time working around subtle incompatibilities between web browsers and platforms, and JavaScript's lack of modularity makes sharing, testing, and reusing AJAX components difficult and fragile. GWT helps developers avoid many of these headaches while offering developers the same dynamic, standards-compliant experience. Developers only need to write their front end in the Java programming language, and the GWT compiler converts the Java classes to browser-compliant JavaScript and HTML [20].

We adopt Google Web Toolkit to build our configuration GUI. The advantage of using GWT is that we can easily create our GUI dynamically. Besides, our configuration tool can separate constructors and users into two places. By building a web server, our configuration GUI becomes naturally a web site, hence every users can do their configuration remotely via internet. In addition, we do not need to care about the portability of configuration GUI made by GWT. All they need to do their configuration via our configuration GUI is just a web browser and do not have to consider whether their platform is adaptable for our system.

4.2.3 Implementation of Element *dependency*

The current implementation of the element *dependency* is presented as an equation. There are two different manifestations defined by the attribute *type* in the element *dependency*. One is the configuration is presented as the element *choice*, and its multiple options are assembled from other configurations whose data type are also *choice*. For example, when defining data cache way: 2 or 4; data cache set: 128 or 256; data cache line size: 32B as shown in fig 3.3 to fig 3.6. Then the available cache size will be presented as the element *choice*, and their multiple options are 8 KB ($2 \times 128 \times 32$), 16 KB ($2 \times 128 \times 32$, $4 \times 128 \times 32$), and 32 KB ($4 \times 256 \times 32$). In this case, we only configure the available cache size but do not configure the other three.

The second manifestation is that the configuration's data is an equation which can involves other configuration whose data type is *value*. After calculating the equation, the configuration is assigned to the computing result and will be presented as the element *value*. For example, the same as above, we can define the available cache size is equal to the product of the data cache way, data cache set, and data cache line size. After we configure the three configurations, the value of available cache size will be generated automatically instead of manually. This manifestation is appropriate to the configuration that is assembled from mass configurations.

Because the element *dependency* is usually presented as an expression, we

seek a parser to obtain the answer. JEP (Java Math Expression Parser) is a Java library for parsing and evaluating mathematical expressions. With this package developers can enter an arbitrary formula as a string, and instantly evaluate it. JEP supports user defined variables, constants, and functions. A number of common mathematical functions and constants are included [14]. By using JEP, we can obtain the answer of an expression rapidly, and we can also save the effort for building our own parser for this purpose.

4.3 Output Layer

After users finished doing their configuration, the output generator in the configuration description layer receives feedbacks from user, and then transfers the output information to the output layer.

In EMCXML, each configuration contains the element *output* which indicates the output information to the output layer. The element *output* consists of uncertain number of element *block* whose function is to store the output information of a specific output target, which described by the element *filename* in *outputfile*. After users do their configuration, the output generator will modify the elements *block*, and send them to the specific targets. For example, if the first element filename in the configuration description file is "foo.bar", then the first element *block* of each configuration will output to "foo.bar". By this design, our configuration tool can output to multiple targets in a sequential

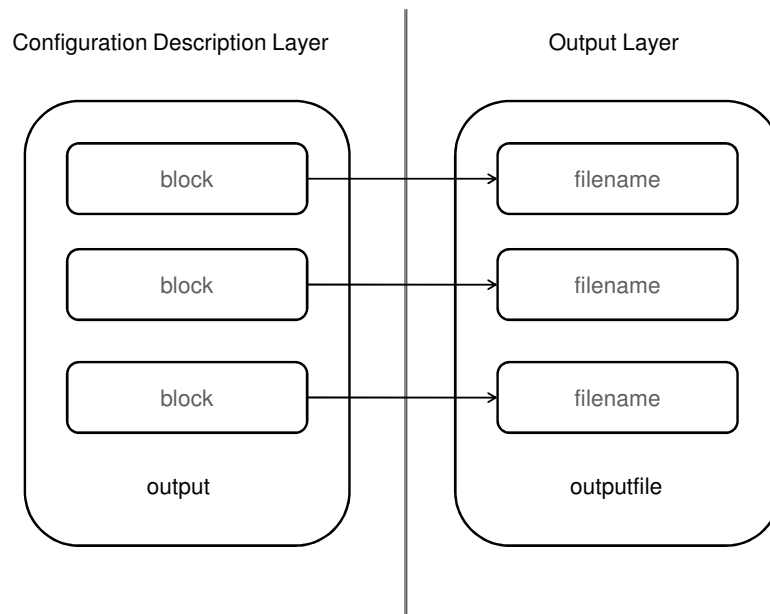


Figure 4.2: Output Information from Configuration Description Layer to Output Layer

series. A special case is if a configuration does not want to output to a specific target, then its corresponding element *block* will be empty. After the output is generated, the EMC Tool can trigger an outer function and do the following jobs automatically. This action can effectively extend the ability of the EMC Tool.

Output layer is also a module as we described before, in other words, every output targets recorded in the element *outputfile* can not only be text files, but also videos or audio files. With the assistance of the output layer, we can do mass works we like. Besides, by adjusting the output layer, its functions are not limited.

Chapter 5

An Example using EMC Tool

In embedded systems, a Board Support Package (BSP) is implementation specific support code for a given board that conforms to a given operating system. It is commonly built with a bootloader that contains the minimal device support to load the operating system and device drivers for all the devices on the board.

We concluded the booting process of Linux on ARM boards as shown in figure 5.1. We classify the processes into four categories:

- On chip functions
- Peripherals
- Related Tools
- Bootstrap

On chip functions describe the hardware specification of the target board, including processor settings, interrupt controllers, memories, etc. Peripherals

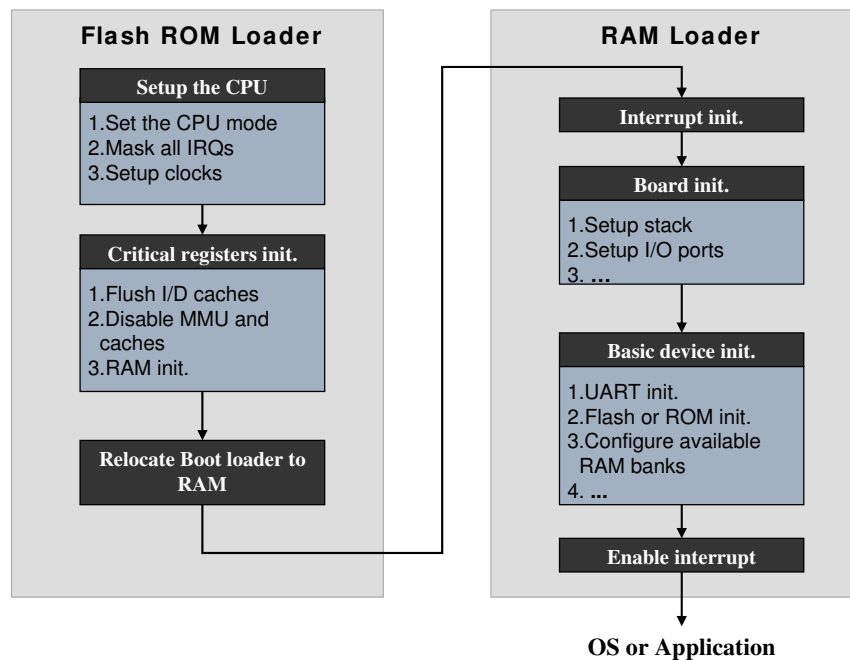


Figure 5.1: Linux Booting Flow

is used to describe the various peripherals of an embedded system, i.e., UART, flash, Ethernet controller, etc. Related tools describe the tool chain of the target board, i.e., cross compiler, debugger, etc. Bootstrap part records the booting process of an operating system, therefore the BSP Tool can generate the appropriate bootstrap code correspondingly.

Our EMC Tool can help developers maintain and configure BSP. The first step is, those four categories above can be described by EMCXML respectively as shown in figure 5.2. Sequentially, developers input the EMCXML files into the EMC Tool, and configure their BSP via the configuration GUI provided by our tool. The three parts, which are on chip functions, peripherals, and bootstrap will become source code by the EMC Tool after developers finish

```

<main>
  <config>
    <name>OnChipFunction</name>
    <display>
      <prompt>On Chip Functions</prompt>
      <description>On Chip Functions</description>
    </display>
    <type>
      <menu></menu>
    </type>
    <output><block></block></output>
    <enable>true</enable>
    <children>
      <config>
        <name>core</name>
        <display>
          <prompt>CPU Core</prompt>
          <description>Settings about CPU core.</description>
        </display>
        <type><menu></menu></type>
        <output><block></block></output>
        <enable>true</enable>
        <children>
          <config>
            <name>SetProcessorModeUsr</name>
            <display>
              <prompt>Set Processor Mode to Usr</prompt>
              <description>Set Processor Mode to Usr</description>
            </display>
            <type>
              <text>
                mrs  r0,cpsr
                bic  r0,r0,#0x1f
                orr  r0,r0,#0x10
              </text>
            </type>
          </config>
        </children>
      </config>
    </children>
  </config>
</main>

```

Figure 5.2: An EMC XML Example of Describing BSP

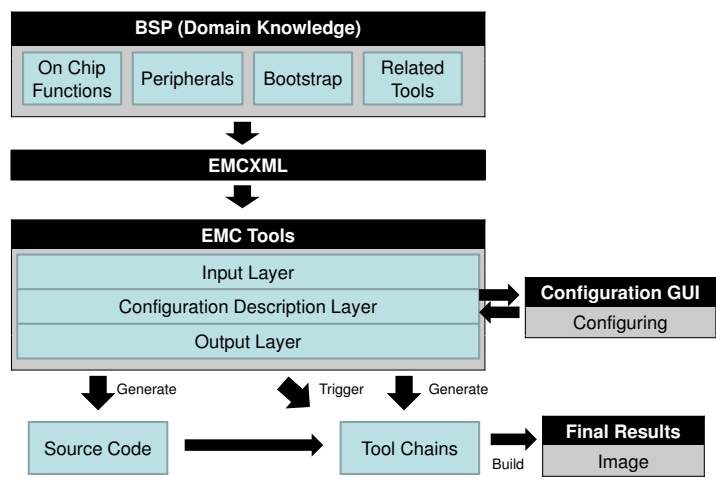


Figure 5.3: Configuring BSP Using EMC Tool

configuring. Developers also configure the related tools, and the EMC Tool will build the tool chains of the target board. Ultimately, the EMC Tool will trigger tool chains after source code is generated completely, and build the desired image automatically. The whole configuration flow is shown in figure 5.3.

Chapter 6


Conclusion and Future Work

In this thesis, we propose an extensible model-based configuration tool to assist people handling their configuration. With our tool, the complexity of configuration problem can be reduced definitely. In order to adapt to different domains, our EMC Tool has no built-in domain knowledge. We define EMCXML to describe the domain knowledge and model each configuration. Compare to other configuration languages, EMCXML has a short learning curve, sound expressiveness and wide extensibility. After defining the domain knowledge of configuration, the EMC Tool is able to parse the EMCXML and create a configuration graphical user interface accordingly. Via the GUI, the developers can easily do their configuration. Besides, the configuration tool has ability to output to distinct multiple targets and trigger outer functions consequently. In addition, the EMC tool is modulized; hence we can modify and adjust any modules in this configuration tool for our sake. For example, we can alter the web GUI to a text-based GUI for text console or alter the output layer so that

the tool can output to video files. According to the modularized structure, our configuration is towards a much wider scenario and scope. Therefore, we believe that configuration can be much easier and efficient with the assistance of our extensible model-based configuration tool. Developers do not need to worry about the complexity of configuration and can handle configuration problems easily.

In current implementation, the dependency of configuration is represented as an expression, and will be improved by represented as script language. By allowing built-in script language, the ability of EMC Tool will be greatly increased. Another issue in current implementation is the concept of constraint is not implemented into our EMC Tool, and We will also implement this feature in the future design. In current design of EMC Tool, we can handle configuration problem by modeling domain knowledge by users' self. However, there are some conditions that are difficult to model domain knowledge. For instances, the domain knowledge involves subjectivity such as the degree of delicious or beauty. In addition to this issue, our configuration tool lacks reusability of configuration models. In future research, we will focus on how to build the configuration models in an obscure condition. Besides, we will also research about how to reuse the components so that the configuration time can be reduced. The last issue is XML document wastes mass disk spaces, and in the future, we will attempt the solution: XML compression.

Bibliography

- 
- [1] Agnar Aamodt and Enric Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications*, 7:39–59, 1994.
- [2] Alexander Borgida, Ronald J. Brachman, Deborah L. McGuinness, and Lori Alperin Resnick. Classic: A structural data model for objects. *ACM SIGMOD Record*, 18:58–67, 1989.
- [3] CML2. <http://catb.org/esr/cml2/>.
- [4] XAmple XML Editor. <http://www.felixgolubov.com/xmleditor/>.
- [5] Peter H. Feiler, David P. Gluch, and John J. Hudak. *The Architecture Analysis & Design Language (AADL): An Introduction*. Society of Automotive Engineers, Feb, 2006.
- [6] Frederick Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28:921–932, 1985.
- [7] Extensible Markup Language (XML) homepage of W3C. <http://www.w3.org/xml/>. 1998.

- [8] Kbuild. <http://kbuild.sourceforge.net/>.
- [9] Apache Software License. <http://xml.apache.org/license>.
- [10] Brett McLaughlin. *Java & XML*. O'Reilly, second edition, 2001.
- [11] Sanjay Mittal and Felix Frayman. Towards a generic model of configuration tasks. *Proc. 11th Int'l Joint Conf. on Artificial Intelligence*, pages 1395–1401, 1989.
- [12] W3C Document Object Model. <http://www.w3.org/dom/>.
- [13] Apache Xerces2 Java Parser. <http://xerces.apache.org/xerces2-j/>.
- [14] Java Math Expression Parser. <http://www.singularsys.com/jep/>.
- [15] Daniel Sabin and Rainer Weigel. Product configuration frameworks - a survey. *IEEE Intelligent Systems*, 13:42–49, August, 1998.
- [16] W3C XML Schema. <http://www.w3.org/xml/schema>.
- [17] John A. Stankovic, Ruiqing Zhu, Ram Poornalingam, and Chenyang Lu. Vest: An aspect-based real-time composition tool. *Real-Time Applications Symposium*, May, 2003.
- [18] J. Tiihonen, T. Soinen, T. Mnnist, and R. Sulonen. State-of-the-practice in product configuration - a survey of 10 cases in the finnish industry. *Knowledge Intensive CAD, First Edition*, 1996.
- [19] Apache Configuration Tool.
<http://www.zecos.com/apache/configuration.html>.

[20] Google Web Toolkit. <http://code.google.com/webtoolkit/>.

