Project N°: **FP7-231620**

Project Acronym: **HATS**

Project Title: **Highly Adaptable and Trustworthy Software using Formal Models**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

Future and Emerging Technologies

Deliverable D3.2
Model Mining

Due date of deliverable: (T36)

Actual submission date: March 1, 2012

Start date of the project: **1st March 2009** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **NR**

Final version

Integrated Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Model Mining

This document summarises deliverable D3.2 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at http://www.hats-project.eu.

This deliverable investigates various algorithms and techniques to build abstract models from concrete inputs in a (partly) automated fashion. Our common name for these techniques is *model mining*. The motivation for model mining is to assist developers in writing models of their code or models describing the domain in which the developers are working. Mined models may then be used for analysis, understanding or reorganising a product line. Our work on model mining builds on and extends results from machine learning, formal specification, and program analysis.

List of Authors

Elvira Albert
Dilian Gurov
Karl Meinke
Bjarte M. Østvold
José Miguel Rojas
Ina Schaefer

Contents

Chapter 1

Introduction

This deliverable reports on three approaches with a common theme: building abstract models from concrete inputs using automated or partly automated techniques. We refer to these techniques collectively as *model mining*. The motivation for model mining is to assist developers in writing models of their code or models describing the domain in which the developers are working. Model mining relies on algorithms and techniques from machine learning, formal specification, and program analysis. These algorithms and techniques allow models to be extracted, automatically or semi-automatically, from other artifacts that developers may already have available, or that they can provide more easily that providing the model itself. The ability to go from code to models contributes to round-trip development of product lines.

1.1 Overview

Here is an overview of the three approaches to models mining.

- *Mining product models from product code* (Chapter 2). This approach is concerned with mining ABS models from Java program in bytecode form, in particular Java program employing a specific distributed message exchange technology (JMS). The techniques applied are static analysis of bytecode programs and exploiting programmer design knowledge via source code annotations. The result from mining is an ABS model of the concrete Java program. A prototype tool, JMS2ABS, exists to do such product mining.

- *Mining variability models from product descriptions* (Chapter 3). This approach is concerned with mining variability models (called feature models in the work task description) from sets of product descriptions. The starting point for mining is a collection of abstract descriptions of individual products—product models. The result of mining, if successful, is a single variability model that describes all of the products in a compact manner and that satisfies certain desirable properties. The kind of variability models studied are general and not tied specifically to products implemented in a particular language. We describe how to make a variability model from a set of Java products; we could in the same manner make a variability model for a set of ABS models.

- *Mining product models from product behaviour* (Chapter 4). Automata learning provides the possibility to infer behavioral models of systems from behavioral data, namely input/output data. For this reason, it is also known as black-box learning. This approach is particularly useful for giving insight into an unknown software implementation of a model. In fact, the approach also is valuable for software testing (shown elsewhere in HATS Work Package 2). A common objection to model-based development is the cost of both developing models, and perhaps more importantly, *revising models*, in a modern agile development environment. Automaton learning can make important contributions to the latter.

  On the other hand, automaton learning is not without complexity problems associated with learning large systems. Even relatively small models may require millions of queries. Furthermore, current

techniques do not extend well to abstract data types or infinite data types. In Chapter 4, we show how some of these problems can be solved. However, improving the performance of learning algorithms is challenging, and our research cannot be considered finished in this respect. Therefore, we have also considered improvements to learning algorithms over finite data types by methods of data abstraction (abstraction using Boolean data types and Boolean learning) and system abstraction, (using both bit slicing methods and incremental learning techniques). This research addresses the *key issue managing the volume and detail of information by the twin principles of abstraction and approximation.*

Note about terminology: In the Description of Work, the text for Work Task 3.2 uses the term feature models (and feature mining). While working on the deliverable we decided that solution-space variability model was a better term for the notion introduced in the task text.

1.2 Summary

In this section we summarize the main chapters of the deliverable.

1.2.1 Extraction of Abstract Behavioural Models

Distributed systems are hard to program, understand and analyze. Two key sources of complexity are the many possible behaviours of a system, arising from the parallel execution of its distributed nodes and the handling of asynchronous messages exchanged between the nodes. We show how to systematically construct *executable models* of publish/subscribe systems based on the Java Messaging Service (JMS). These models, written in the executable Abstract Behavioural Specification (ABS) language, capture the essential parts of the messaging behaviour of the original Java systems, and eliminate details not related to distribution and messages. We report on JMS2ABS, a tool that automatically extracts ABS models from the *bytecode* of JMS systems. Since the extracted models are formal and executable, they allow us to reason about the modeled JMS systems by means of tools built specifically for the modeling language. For example, we have succeeded to apply simulation, termination and resource analysis tools developed for ABS to, respectively, execute, prove termination and infer the resource consumption of the original JMS applications.

This work was carried out by NR and UPM.

1.2.2 Simple Hierarchical Variability Models

A key challenge in software product line engineering is to represent solution space variability in an economic, yet easily understandable fashion. We introduce the notion of a hierarchical variability model to specify a family of products. In this model, a family is represented by a *common* set of artifacts and a set of *variation points* with associated variants. A variant is again a hierarchical variability model, leading to a hierarchical structure. However, hierarchical variability models are not unique with respect to the families they define. Hence, we propose a quantitative measure on hierarchical variability models that expresses the degree to which a variability model captures commonality and variability in a family. By imposing well-formedness constraints, we identify a class of variability models that, by construction, have maximal measure and are unique for the families they define. For this class of *simple families*, we provide a procedure that reconstructs their hierarchical variability model. We illustrate the approach by a small product line of Java classes. The simplicity of the proposed variability model and its characterization make it a suitable starting point to investigate and compare more general hierarchical variability models.

This work was carried out by KTH, NR and CTH[1].

[1] TU Braunschweig is the current affiliation of Ina Schaefer; she was formerly with CTH.

1.2.3 Symbolic Inference of Automata Models

Most component-based systems in use today lack adequate documentation and make use of un/under-specified components. In fact, the popular component-based software design paradigm naturally leads to under-specified systems, as most libraries only provide very partial specifications of their components. Moreover, typically, revisions and last minute changes hardly enter the system documentation.

Machine learning has been proposed to overcome this situation by automatically 'mining' system models and then updating the required documentation. Promising results have been obtained here, for example in the context of embedded systems, by using active automata learning technology. There is also a high potential to exploit other machine learning techniques such as symbolic learning and inductive inference.

These approach includes research on *computational learning techniques to dynamically infer automata models* (including Kripke structures and Mealy machines) based on dynamic black-box observation of system behaviours. The results of this research have been successfully applied elsewhere in Work Task 2.3 to the problem of black-box automated test case generation. Here, we will focus more on reporting the model-inference algorithms themselves.

This work was carried out by KTH.

1.3 List of Papers Comprising Deliverable D3.2

This section lists all the papers that comprise this deliverable, indicates where they were published, and explains how each paper is related to the main text of this deliverable. In addition, two papers, each related to one of Chapters 2 and 3, have been submitted for publication.

As requested by the reviewers, the papers are not directly attached to Deliverable D3.2, but are made available on the HATS web site at the following URL: http://www.hats-project.eu/sites/default/files/D3.2. Direct links are also provided for each paper listed below.

Paper 1: CGE: a Sequential Learning Algorithm for Mealy Automata,

This paper [42] introduces a symbolic learning algorithm for Mealy machines computing over an abstract data type modeled as a many-sorted algebraic structure. The paper proves that the algorithm correctly learns in the limit.

This paper was written by K. Meinke and was published in the proceedings of ICGI the tenth annual Colloquium on Grammatical Inference. (ICGI 2010)

Download Paper 1.

Paper 2: Learning-Based Testing for Reactive Systems using Term Rewriting Technology

This paper [43] shows how the CGE learning algorithm of paper 1 above can be applied to black-box specification-based software testing. The paper introduces an architecture for testing, based on model checking by term rewriting. A case study of learning and testing the TCP/IP protocol is considered.

This paper was written by K. Meinke and F. Niu and was published in the proceedings of the twenty third IFIP International Conference on Testing Software and Systems (ICTSS 2011).

Download Paper 2.

Paper 3: Correctness and Performance of an Incremental Learning Algorithm for Finite Automata

This paper [44] presents an incremental learning algorithm for learning DFA. A correctness proof of the algorithm is given. This paper represents our first attempt to consider incremental learning, where a hypothesis automaton is incrementally generated using only the currently available behavioral information.

This represents an important model abstraction technique similar to dynamic program slicing, and is needed for efficient learning of large models.

This paper was written by K. Meinke and M. Sindhu and was presented as a poster at the third Asian Conference on Machine Learning (ACML 2011).

Download Paper 3.

Paper 4: Incremental Learning-Based testing for Reactive Systems

This paper [45] presents an incremental learning algorithm for learning deterministic Kripke structures. Such structures can encode computations over any finite data type. The algorithm generalises the DFA learning algorithm of paper 3 above by applying bit-slicing and lazy partition refinement. A correctness proof of the algorithm is given. The learning algorithm is evaluated in the context of model-based black-box software testing. The performance of learning and testing for two case studies of a cruise controller and an elevator model are considered.

This paper was written by K. Meinke and M. Sindhu and was presented at the International Conference on Tests and Proofs (TAP2011).

Download Paper 4.

Chapter 2

Extraction of Abstract Behavioural Models

2.1 Introduction

Reverse engineering is a key technique to help debugging, understanding, managing, refactoring, and generally improving software that is available only in executable form. In this chapter we focus on reverse engineering/decompilation of distributed Java applications given in bytecode form. We study a specific class of publish/subscribe systems [25] built using the Java Messaging Service (JMS) [34]. Our goal is to extract *abstract behavioural specifications* that capture the essentials of the messaging behaviour, eliding implementation details, but preserving enough behaviour that analysis can draw conclusions about distribution and resource consumption of the original systems. The modelling language, that is, the abstract behavioural specification language (ABS), has a *functional sub-language* which allows abstracting from implementation details: abstract data types and functions are used to specify internal, sequential computations, while concurrency and distribution are handled in the imperative object-oriented language. The ABS has a formal operational semantics [36] and its analysis is supported by the HATS tool chain

We report on JMS2ABS, a tool which automatically extracts an ABS model from a JMS application in bytecode form. In languages such as Java and C#, reasoning on bytecode is possible even when source code is not available. The main phases of the extraction process performed by JMS2ABS are the following: (1) We decompile the bytecode into a higher-level intermediate representation which, in contrast to the original bytecode, features structured control flow, explicitly represents the JVM operand-stack by means of variables, and has some other features which later make the extraction of models feasible. (2) The user optionally adds annotations which indicate which methods of the code should be transformed into functions and which ones into imperative methods. The lack of annotations implies that the model will be purely imperative. Based on the user-provided annotations, the tool takes the intermediate representation and generates either functions, with associated abstract data types, or methods defined inside classes. (3) The implementation of publish/subscribe JMS application relies on a series of library methods (often called *middleware*) which have been modelled in ABS separately. This means that, when the extraction procedure finds a call to one these library methods, it just inserts a call to the corresponding pre-defined ABS function or method. The main contributions of our work can be summarized as follows:

- Section 2.3 provides a general and system-independent model of the core part of JMS publish/subscribe systems;
- In Section 2.4, we define a procedure for translating the code of a JMS publish/subscribe system into an executable model, and realise this as a tool;
- Section 2.5 applies existing tools developed for the ABS language in order to draw conclusions about the systems;

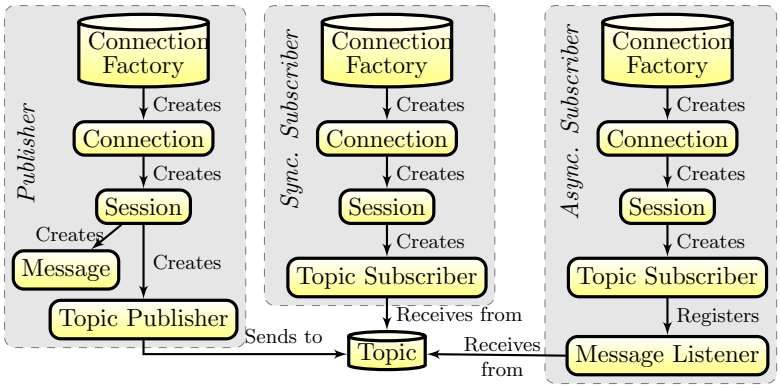

Figure 2.1: Overview of the JMS publish/subscribe programming model.

- Finally, Section 2.6 reports on a prototype implementation of our approach and evaluates it on two JMS examples.

2.2 Publish/Subscribe Communication in JMS

JMS is a standard for message communication in enterprise systems [34]. It is a *library-based* approach for developing distributed applications, offering API methods for configuring message passing services and for performing the message passing (i.e., encode, send, receive, and decode messages in such systems). One may realise various kinds of messaging systems using JMS, among them, we focus on publish/subscribe systems.

The *publish/subscribe* communication model is receiving increased attention because it provides a loosely coupled form of interaction which is useful in large-scale distributed systems. Fig. 2.1 provides an overview of the publish/subscribe programming model as it is implemented in JMS. Here *subscribers* have the ability to express their interest in a (pattern of) events or messages in order to be later notified of any message generated by the *publisher* that matches their registered interest. The basic model for publish/subscribe interaction relies on a message notification service, or *middleware*, providing storage and management for subscriptions, mediation and decoupling between publishers and subscribers, and efficient messages delivery. Such middleware manages addressable message destination objects denoted as *topic*. The steps that *publishers* and *subscribers* perform, as depicted in Fig. 2.1, are: (1) Discover and join a topic of interest by means of a *connection factory* of topics. (2) Establish a *connection* to the factory and then start a new *session* for a such connection. (3) Create a *topic subscriber* for the session which allows receiving (subscribers) and sending (publishers) messages related to the topic of interest. (4) Create and publish a message (publisher). (5) Receive a message (subscriber).

We consider the subset of JMS components depicted in Fig. 2.1, capturing the essence of the publish/-subscribe communication model. In addition, a model of a JMS system must include the state information and logic that decides how messages are processed and exchanged. Features such as transactions or failure recovery are outside the scope of this chapter. Fig. 2.2 shows an example of a JMS publish/subscribe implementation of a basic fruit supply business model, consisting of a `FruitSupplier` class that acts as a publisher of updates for topic `"PriceLists"`; `AsynchSuperMarket` class implements asynchronous updates receipts from the topic, time-decoupled (i.e., non-blocking) from the publisher; and `Example` class provides the `main` method that initializes instances of `FruitSupplier` and `AsynchSuperMarket`.

Note that the different components are created and retrieved by invoking API methods. In particular, `ConnectionFactory` and `Topic` objects can be either created dynamically or found using JNDI services[1]. Subscribers can retrieve messages either asynchronously using a `MessageListener` object or synchronously through the (blocking) `receive` method of a `TopicSubscriber` object.

[1] http://www.oracle.com/technetwork/java/jndi/

```
class FruitSupplier extends Thread {
  void run() {
    fac = new TopicConnectionFactory();
    con = fac.createTopicConnection();
    ses = con.createTopicSession(...);
    topic = ses.createTopic("PriceLists");
    publisher = ses.createPublisher(topic);
    message = ses.createObjectMessage(priceList);
    publisher.publish(message);//execution continues
    con.close(); } }
class AsynchSuperMarket extends Thread {
  PriceList priceList;
  class PriceListListener implements MessageListener {
    void onMessage(ObjectMessage m) {
      newPriceList = m.getObject();
      updatePrices(newPriceList); }
    void updatePrices(PriceList l) {
      Product p;
      for (int i = 1; i <= l.length(); i++) {
        p = l.get(i);
        if (priceList.contains(p)) priceList.update(p);
        else priceList.insert(p); } } }
  void run() {
    fac = new TopicConnectionFactory();
    con = fac.createTopicConnection();
    ses = con.createTopicSession(...);
    topic = ses.createTopic("PriceLists");
    subsc = topicSession.createSubscriber(topic);
    listener = new PriceListListener();
    subsc.setMessageListener(listener);
    con.start();//execution continues
    con.close(); } }
class Example {
  void main(...) { new AsynchSuperMarket().start();
                   new FruitSupplier().start(); } }
```

Figure 2.2: Excerpt of implementation of publish/subscribe in JMS (Java code simplified to save space).

2.3 Modeling Publish/Subscribe Systems in ABS

This section shows how to model the behaviour of a publish/subscribe system implemented using JMS by means of the ABS language. The model will abstract away implementation-related details of the Java distributed application while still capture the essence of cooperation among the components of the system. Our goal is to preserve all essential properties concerning distribution and performance but improve on clarity and tractability for automatic analysis purposes. Our starting point is the JMS system of Fig. 2.1, whose model in ABS is shown in Fig. 2.4 (in the model, *updatePrices* is a function that will be defined later). In particular, we focus on the components that participate in the system (Sec. 2.3.1) and the operations that can be executed (Sec. 2.3.2). Sec. 2.4 will then describe how to automate the model extraction process.

2.3.1 Distributed Entities

Our ABS model creates only one concurrent object per participant in the distributed communication, namely publishers, subscribers and the middleware; see Fig. 2.3. Thus, each concurrent entity in ABS encapsulates the behaviour of several JMS components that will communicate with the remaining entities by means of asynchronous calls and future variables.

Clients: Publishers and Subscribers. As we have seen in Sec. 2.2, a JMS system relies on a number of objects in order to perform any distributed operation. This design makes JMS portable and interoperable across multiple messaging products. However, it often makes the resulting programs harder to understand and thus analyze. Instead of relying on such object machinery, and aiming not to lose generality, we propose

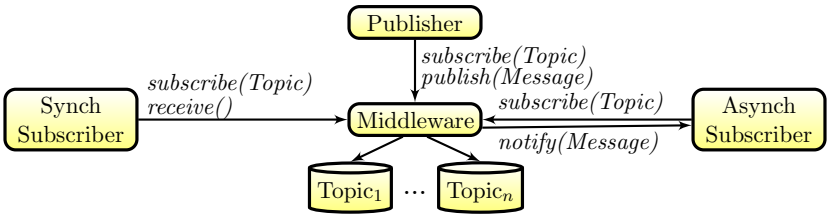

Figure 2.3: Concurrent entities in ABS models (counterpart of Figure 2.1 for JMS).

JMS instructions	**equivalent ABS models**
Create an object or a distributed object	
`obj = new C();` `//if class C implements Thread`	`obj = new C();` `obj = new cog C();`
Establish new session	

`f = new TopicConnectionFactory();`
`c = f.createTopicConnection();`
`s = c.createTopicSession(...);`
`t = new Topic("TopicName");`
	`s = middleware.createSession();`
	`s = middleware.createTopic("TopicName");`

Send a message	

`pub = s.createPublisher(t);`
`connection.start();`
`m = s.createTextMessage();`
`m.setText("message");`
`pub.publish(m);`
	`pub = s.createPublisher(t);`
	`m = "message";`
	`pub!publish(m);`

Receive a message synchronously	

`sub = s.createSubscriber(t);`
`connection.start();`
`m = topicSubscriber.receive();`
	`sub = s.createSubscriber(t);`
	`Fut<Message> f = s!receive();`
	`message = f.get;`

Receive a message asynchronously	

`sub = s.createSubscriber(t);`
`l = new TextListener();`
`sub.setMessageListener(l);`
`connection.start();`
	`sub = s.createSubscriber(t);`
	`l = new MessageListener();`
	`sub.setMessageListener(l);`

Table 2.1: Example mapping from Java/JMS to ABS.

an abstraction mechanism in ABS based on a more compact set of classes and interfaces to implement publisher and subscriber clients. Namely, a publish/subscribe client in ABS will just need to create a session and a publisher or subscriber object to interact with the middleware.

Middleware. In a publish/subscribe system, a highly distributed entity called the Message Oriented Middleware (MOM) is responsible for routing messages from publishers to the right subscribers. In our model of JMS we use a centralized logical view of the middleware entity. Even if a centralized implementation is mostly undesirable in the implementation of a real distributed system, for the purpose of modeling and analyzing JMS publish/subscribe systems, it is adequate. The middleware entity relies on the concurrency model of ABS to provide publish/subscribe services. Its internal routines are transparent to clients, but exploitable for automatic analysis tools. The `main` block of the ABS model creates the initial configuration of the publish/subscribe system, see Line 23 of Fig. 2.4 (the counterpart of Lines 34–35 of Fig. 2.2). Observe that the model uses COGs to represent each of the distributed/concurrent entities.

2.3.2 Operations

Here we consider how to model the operations of a publish/subscribe system: *subscribe*, *unsubscribe*, *publish*, and *notify*.

```
1  class FruitSupplier(Middleware mw) {
     Unit run() {
3      Topic topic = "PriceLists";
       TopicSession session = mw.createSession();
5      TopicPublisher publisher = session.createPublisher(topic);
       ObjectMessage message = new ObjectMessage(priceList);
7      session!publish(message);
     } }
9  class MessageListener() {
     Unit onMessage(ObjectMessage m) {
11     newPriceList = m.getObject();
       priceList = updatePrices(newPriceList);
13   }
   }
15 class AsynchSuperMarket(Middleware mw) {
     Unit run() {
17     Topic topic = "PriceLists";
       TopicSession session = mw.createSession();
19     TopicSubscriber subscriber = session.createSubscriber(topic);
       Listener listener = new MessageListener();
21     subscriber!setMessageListener(listener);
     }
23 { Middleware mw = new cog Middleware();
     new cog AsynchSuperMarket(mw); new cog FruitSupplier(mw);
25 } //main block
```

Figure 2.4: Extracted ABS model for the running example. (Middleware model not shown.)

Message Sending. A publisher sends a message to the topic using a session, see method `run` of class `FruitSupplier` (Lines 3–10 of Fig. 2.2). The asynchronous semantics of the operation can be simulated by an ABS asynchronous method call, see Lines 3–8 of Fig. 2.4. In JMS, the sending operation implies some decisions regarding delivery mode, priority and time-to-live for the message. These configuration parameters can be global to a message publisher or specific for each message. For flexibility, we use the latter option and include configuration parameters as properties of messages.

Asynchronous Message Receipt. Method `run` of class `AsynchSuperMarket` in Fig. 2.2 shows that asynchronous message receipt in JMS is achieved by instantiating the `MessageListener` class. The new object is bound to the subscriber object and is able to receive and process incoming messages in its `onMessage` method (named `notify` in the publish/subscribe literature [25]). This method is triggered from the JMS provider upon arrival of a new message to the topic (Lines 24–32 of Fig. 2.2). In ABS, an equivalent asynchronous message receipt is implemented in Lines 17–22 of Fig. 2.4. The concurrent behaviour, i.e., the interaction with different topics simultaneously, is achieved by sharing the session object among clients within the same COG. A serial order of outgoing and incoming messages is implicitly modelled when using a shared session object. Table 2.1 summarizes the mappings that we have described along this section for our particular example.

2.4 Automatic Extraction of ABS Models from JMS

Figure 2.5 provides an overview of the main steps performed by JMS2ABS for automatically extracting ABS models from JMS publish/subscribe systems. The tool receives as input the *bytecode* associated to the JMS publish/subscribe system and, optionally, a set of *annotations* which indicate which methods of the code should be transformed into functions and which ones into imperative methods. The lack of annotations leads to a purely imperative translation. Intuitively, the following stages are carried out by the extraction process. First, the bytecode is decompiled into a higher-level intermediate representation (IR) which, among other things, features structured control flow. Then, a driver module reads the IR and the set of annotations and directs the model extraction process either towards a functional implementation or towards an imperative one. The extraction of functional code requires a static single assignment (SSA) transformation [10] and

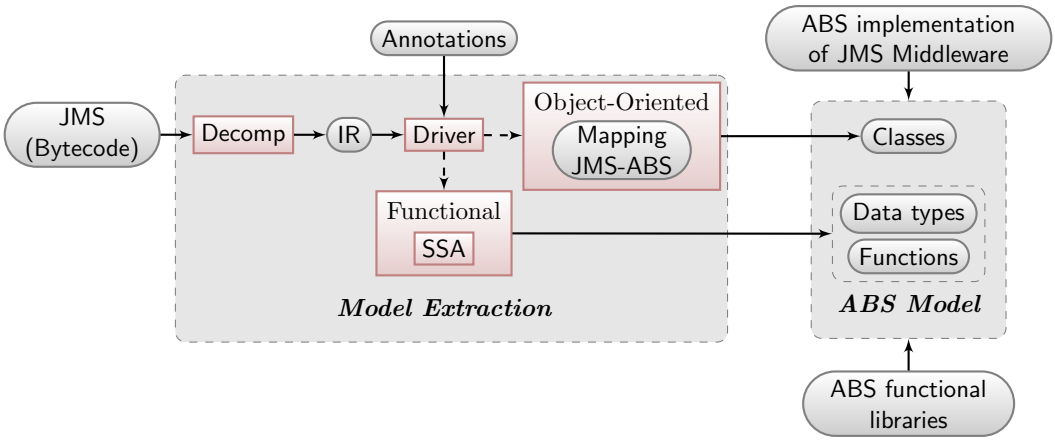

Figure 2.5: Overview of main components of JMS2ABS.

automatically generates functional data types and functions. The ABS library functions include standard data types for lists, sets, etc. and some common functions on these types. They are used by the translation when possible. The imperative object-oriented extraction is based on the modelling of JMS using ABS defined in Table 2.1. As an external component to this process, we have available the ABS implementation of the specific middleware implementation in use. As a result of the process, an ABS model is obtained which includes abstract data types, functions and classes. The following sections describe in detail the main components of JMS2ABS.

2.4.1 From Bytecode to Intermediate Representation

A method m in a Java (bytecode) program is represented by a set of *procedures* in the IR such that there is an entry procedure named m and the remaining ones are intermediate procedures invoked only from m. The translation of a program into the IR works by first building the control flow graph (CFG) from the program, and then representing each block of the CFG in the IR as a rule. The process is identical to that of Albert *et al.* [4], hence, we will not go into the details of the transformation but just show the syntax of the transformed program. A *program* in the IR consists of a set of *procedures* which are defined as a set of (recursive) rules. A procedure p is defined by a set of *guarded rules* which adhere to the following grammar:

$$\begin{array}{rcl@{\qquad}rcl} \mathit{rule} & ::= & p(\bar{x},\bar{y}) \leftarrow g, b_1, \ldots, b_n & g & ::= & \mathit{true} \mid \mathit{exp}_1\ \mathit{op}\ \mathit{exp}_2 \mid \mathsf{type}(x, C) \\ \mathit{exp} & ::= & x \mid \mathsf{null} \mid n \mid x{-}y \mid x{+}y \mid x{*}y & \mathit{op} & ::= & > \mid < \mid \leq \mid \geq \mid = \mid \neq \\ b & ::= & x{:=}\mathit{exp} \mid x{:=}\mathsf{new}\ c \mid x{:=}y.f \mid x.f{:=}y \mid q(\bar{x},\bar{y}) & & & \end{array}$$

where $p(\bar{x},\bar{y})$ is the *head* of the rule; $\bar{x}$ (resp. $\bar{y}$) are the input (resp. output) parameters; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables; f a field name, and $q(\bar{x},\bar{y})$ a procedure call by value. The language supports class definition and includes instructions for object creation, field manipulation, and type comparison through the instruction $\mathsf{type}(x, C)$, which succeeds if the runtime class of x is exactly C. A class C is a finite set of typed field names, where the type can be integer or a class name. The key features of this representation which simplify the transformation later are: (1) input and output parameters are explicit variables of rules, (2) *recursion* is the only iterative mechanism, (3) *guards* are the only form of conditional and (4) objects can be regarded as records, and the behavior induced by dynamic dispatch is compiled into *dispatch* rules guarded by a type check.

Example 1. *As an example, let us consider method* `updatePrices` *in Fig. 2.2. The left-hand column of Fig. 2.6 shows the bytecode of this method (which is the input to* JMS2ABS*) and the right-hand column contains the IR that* JMS2ABS *uses which features the three first points above. We can observe that instructions in the IR have an almost one-to-one correspondence with bytecode instructions (rule_7 in the IR corresponds to the CFG block starting at bytecode instruction 7, for example), but they contain as explicit parameters the*

0: iconst_1	*updatePrices*([this,l],[]) ← i := 1,
1: istore_3	*rule_2*([this,l,i],[]).
2: iload_3	*rule_2*([this,l,i],[]) ←
3: aload_1	***length***([l],[s_1]),
4: invokevirtual length:()I	*rule_7*([this,l,i,s_1],[]).
7: if_icmpgt 43	*rule_7*$_1$([this,l,i,s_1],[]) ←
10: aload_1	i ≤ s_1,
11: iload_3	***get***([l,i],[p]),
12: invokevirtual get:(I)LProduct;	***exists***([this,p],[s_2]),
15: astore_2	*rule_21*([this,l,p,i,s_2],[i_p]).
16: aload_0	*rule_7*$_2$([this,l,i,s_1],[]) ←
17: aload_2	i > s_1.
18: invokevirtual exists:(LProduct;)Z	
21: ifeq 32	*rule_21*$_1$([this,l,p,i,s_1],[]) ←
24: aload_0	s_1 = 0,
25: aload_2	***add***([this,p],[]),
26: invokevirtual updatePrice:(LProduct;)V	*rule_37*([this,l,i],[]).
29: goto 37	*rule_21*$_2$([this,l,p,i,s_1],[]) ←
32: aload_0	s_1 ≠ 0,
33: aload_2	***updatePrice***([this,p],[]),
34: invokevirtual add:(LProduct;)V	*rule_37*([this,l,i],[]).
37: iinc 3, 1	*rule_37*([this,l,i],[]) ←
40: goto 2	i_p := i +1,
43: return	*rule_2*([this,l,i_p],[]).

Figure 2.6: Pretty-printed IR for method `updatePrices` of class `PriceList`.

variables on which they operate (the operand stack is represented by means of variables). Another important aspect of the IR is that unstructured control flow of bytecode (i.e., the use of `goto` *statements) is transformed into recursion and loop conditions become* ***`guards`****, as in rules rule_2 and rule_37 for instance.*

2.4.2 From IR to Functions

It is rather simple to transform a set of procedures in the IR into a functional program. We assume that the annotations which indicate which fragments of the code have to be transformed into functions are available and sound (i.e., the code to be transformed is sequential and does not write on the shared data or heap). It would be possible nevertheless that JMS2ABS checks that the soundness conditions are satisfied and throw an exception otherwise. The generation of functions involves three main steps:

- It first performs an SSA transformation on the IR which guarantees that variables are only written once [10].
- For each recursive procedure in the IR, it then generates an associated function with the same name, where each instruction is transformed into an equivalent one in the functional language using ABS's `let` expressions. The process is similar to decompilation of bytecode to a simply typed functional language [37], to TRS [50] or to CLP programs [30]. Hence, we do not go into the details of the process but rather show an example.
- Besides, JMS2ABS has to generate definitions of the data types involved in the functions. This is done by recursively inspecting the types of the class fields until reaching a basic type, and using tuples to group the fields that form an object. For instance, a class `A` which has two fields of types `B` and `C` which, in turn, each have one field of type *Integer*, gives rise to the functional data types `type B = Int; type C = Int; data A = EmptyA | ConsA(B,C);`.

Example 2. *The following function corresponds to the bytecode in Fig. 2.6. It is extracted from the above IR in a fully automatic way. Observe that each recursive procedure in the IR is translated into a function. As usual,* let *expressions are used to perform variable bindings, and* case *expressions to represent*

conditional statements in the original program. Moreover, observe that several data types declarations have been generated from class `PriceList`*. The new algebraic data type* `PriceList` *has two data constructors: one for the empty list (*`EmptyPriceList`*) and one for the combination of a product and another list (*`ConsPriceList(Product,PriceList)`*).*

```
1 //Data type declarations
  type ProductID = Int;  type Price = Int;
3 data Product = EmptyProduct | ConsProduct(ProductID,Price);
  data PriceList = EmptyPriceList | ConsPriceList(Product,PriceList);
5 //Function definitions
  PriceList updatePrices(PriceList l) {
7     let Int i = 1 in loop(priceList,newPriceList,i);
  }
9 PriceList loop(PriceList l1, PriceList l2, Int i) {
    let n = length(l2) in
11    case i <= n of
        true: let p = get(l2,i) in
13                case contains(l2,p) of
                    true: return loop(update(l1,p),l2,i+1);
15                  false: return loop(add(l1,p),l2,i+1);
        false: return l1;
17 }
```

2.4.3 From IR to ABS

All procedures which have not been transformed into functions will become methods of the ABS models. Each ABS class will have as attributes the same ones as in the original Java program. Then, the translation of each method is performed by mapping each instruction in the IR into an equivalent one in ABS. The instructions which involve the distribution aspects of the application are translated by relying on the mapping of Table 2.1.

Example 3. *Fig. 2.7 shows the IR for the Java method* `AsynchSuperMarket.run`*. Observe how instructions in lines 2–5 match with the pattern for session establishment shown in Table 2.1. As a variant of this pattern, the* `Topic` *and* `TopicConnectionFactory` *objects could also be existing instances retrieved via a naming and directory interface (*`jndiLookup` *calls). Instructions in lines 7–9 correspond to the asynchronous receiving of a message.*

```
0  run([this],[])  ←  tConFac := null, tCon := null, tSes := null, topic := null,
1                     tSubscriber := null, tListener := null,
2                     tConFac := new TopicConnectionFactory,
3                     createTopicConnection([tConFac],[tCon]),
4                     createTopicSession([tCon],[tSes]),
5                     createTopic([tSes,this.topicName],[topic])
6                     createSubscriber([tSes,topic],[tSubscriber]),
7                     tListener := new PriceListener,
8                     setMessageListener([tSubscriber,tListener],[]),
9                     start([tCon],[]), close([tCon],[]).
```

Figure 2.7: Pretty-printed IR for method `run` of class `AsynchSuperMarket`.

From the above IR it is straightforward to extract the model for method `run` showed in Fig. 2.4. Because of the correspondence between the involved operations in JMS and ABS, the main properties of the JMS systems (e.g., those regarding *reliability* and *safety* [12]) are clearly preserved.

2.5 Using the ABS Toolset on the Extracted Models

The goal of the extraction of ABS models from bytecode systems is to be able to perform machine analysis of JMS systems via their equivalent ABS models. This section outlines the application of two ABS tools:

the simulator [36] and the COSTABS termination and resource usage analyzer [3].

2.5.1 Simulation

Once compiled, ABS models can be run in a simulator. The ABS toolset has two main simulators, with corresponding back-ends in the ABS compiler: One simulator is defined using rewriting logic and the Maude system [16], and the other is written in Java. The Maude simulator allows modellers to explore the model's state-space declaratively and model check it. The Java simulator does source-level simulation, meaning that modellers can follow the model's control flow at the statement level and observe object or method state. Process scheduling is non-deterministic in ABS. However, both simulators allow modellers to control scheduling of methods. As an example, a modeller can control when a JMS message is sent and when it is received.

2.5.2 Resource and Termination Analysis

Resource analysis [62] (a.k.a. cost analysis) aims at automatically inferring bounds on the resource consumption of programs statically, i.e., without having to execute the program. The inferred bounds are symbolic expressions given as functions of its *input data sizes*. For instance, given a method `void traverse(List l)`, an upper bound (e.g., for number of execution steps) can be an expression on the form *l*200+10*, where *l* refers to the size of the list `l`. The analysis guarantees that the number of steps of executing `traverse` will never exceed such amount inferred by analysis. COSTABS [3], a COSt and Termination analyzer for ABS, is a system able to prove termination and obtain *resource usage bounds* for both the imperative and functional fragments of ABS programs. The resources that COSTABS can infer include termination, number of execution steps, memory consumption, number of asynchronous calls, among others. Knowledge on the number of asynchronous calls is useful to understand and optimize the distributed behaviour of the application (e.g., to detect bottlenecks when one object is receiving a large amount of asynchronous calls).

Example 4. *Let us analyze the resource consumption of the two methods in class* `AsynchSuperMarket` *of the extracted ABS model in Fig. 2.4. COSTABS allows using* asymptotic *(i.e., big O complexity) notation for the results of the analysis and obtain simpler cost expressions. In particular, COSTABS computes the following asymptotic results for the above cost models:*

Method	*#Instructions*	*Memory*	*#Async Calls*
`run`	`max`*(allSubscribers)*	`max`*(allSubscribers)*	*1*
`onMessage`	*m*(m+*`max`*(priceList))+m^2*	`max`*(priceList)*	*1*

The upper bound on the number of instructions inferred for method `run` *depends on the number of clients that are subscribed to the topic (field* `allSubscribers` *of class* `TopicSession`*).* `max`*(f) denotes the maximum value that field f can take. This is because in our current implementation the size of the list of subscribers is not statically known, as it is updated when a new subscriber arrives (the analysis uses* `max`*(allSubscribers) to bound its size). As regards the analysis of* `onMessage`*, it requires analyzing* `updatePrices` *which traverses the new list of prices* `priceList` *and, for each of its elements, it checks whether it already exists or must be added to the local list of prices. The latter requires inspecting the object message* `m` *which is an input parameter of the method. Hence, we obtain a quadratic complexity on the sizes of* `m` *and* `priceList`*. The memory allocation accounts for the creation of the functional data structures. Namely, in method* `run` *(resp.* `onMessage`*), we create the data structure* `allSubscribers` *(resp.* `PriceList`*). Finally, it can be observed that both methods perform a constant number of asynchronous method calls, hence the rightmost column shows a constant complexity (denoted by* 1*).*

A main novelty of COSTABS, which is not available in other systems, is the notion of *cost centers*. This is motivated by the fact that distribution does not match well with the traditional monolithic notion of cost which aggregates the cost of all distributed components together. Albert *et al.* [3] propose the use of cost centers to keep the resource consumption of the different distributed components separate.

Example 5. *In Ex. 4, the cost bound is computed as a monolithic expression which accumulates the resources consumed by all objects together. More interestingly, COSTABS can show the results separated by cost centers. In particular, we consider that all objects of the same class belong to the same cost center (i.e., the share the processor). Now, the execution of method* `AsynchSuperMarket.run` *performs steps in three cost centers, namely in* `AsynchSuperMarket`, `Middleware` *and in* `TopicSubscriber`*. By enabling the cost centers option, COSTABS shows that* `max`*(allSubscribers) is the upper bound on both number of instructions and memory in the cost center* `Middleware`*. In cost center* `TopicSubscriber`*, the upper bounds on number of instructions and on memory consumption are constant. Also, in cost center* `AsynchSuperMarket`*, the upper bound for both cost models is constant. Method* `onMessage` *is integrally executed in the* `AsynchSuperMarket` *cost center (hence the same results of Ex. 4 are obtained).*

Performing cost analysis of a distributed system, using cost centers, allows detecting bottlenecks if one distributed component (cost center) has a large resource consumption while siblings are idle most of the time.

2.6 Prototype Implementation

JMS2ABS can be used on 32-bit Linux systems through a command-line interface. An Eclipse plugin is under development. The system is open-source and can be downloaded from http://tools.hats-project.eu/, together with examples, documentation, etc. Also, available from the same place, is our ABS model of JMS middleware and two examples of how to write publish/subscribe ABS models using the middleware model: One is the running example of the chapter, and the other is a Chat client. Furthermore, we include two Java/JMS example applications from which models may be extracted. Again, one is a the running example of the chapter and the other a Chat example, borrowed from a book on JMS [55] and slightly simplified. The Java code is accompanied by the necessary Java/JMS libraries and a makefile which may be used to run the tool on the Java examples. Although still a research prototype, JMS2ABS is reasonably efficient. For instance, on an Intel(R) Core(TM) i5 CPU at 1.7GHz with 4GB of RAM running Ubuntu Linux 11.10, the overall time to extract the model for the running example is 910 msec. This time is divided into the time for building the CFG (240 msec.), generating and optimizing the intermediate representation (including SSA transformation) (40 msec.) and building and refining the ABS model (630 msec.). The Chat example is smaller and has an overall model extraction time of 790 msec. In this case, the most costly phase is also the model generation and refinements, which takes 490 msec. of the overall time.

2.7 Related Work

Reverse engineering higher-level specifications from complex third party or legacy code has applications in analyzing, documenting and improving the code. Broadly speaking, we can classify reverse engineering tools into two categories: (1) When the higher-level specification is some sort of software visualization formalism which abstracts away most of the program semantics (e.g., UML class diagrams, control flow graphs or variable data flow graphs), reverse engineering is usually applied in order to understand the structure of the source code faster and more accurately. This in turn can detect problems related to the design of the application, to task interactions, etc. (2) When the higher-level specification provides an abstraction of the program semantics, but still the properties of interest are observable on it, reverse engineering can be used to develop analysis tools that reason about the original code by means of analyzing the reverse engineered specification. This has the advantage that, instead of analyzing the complex original code, we develop the tools on a simpler formalism which allows inferring the properties of interest more easily.

Our work falls into the second category. The overall motivation behind our work is to be able to analyze (complex) distributed Java JMS applications by means of tools developed for (simpler) ABS models. In particular, we have been able to apply simulation and *cost analysis* techniques developed for ABS programs [5, 1] to reason on JMS applications. It is widely recognized that publish/subscribe systems are difficult to reason about, and there are several previous approaches to modelling their behaviour using different

formalisms. Baldoni et al. [12] provide one of the first formal computational frameworks for modelling publish/subscribe systems. The focus in this work is different from ours; their main concern is the notion of time in the communication, which allows them to evaluate the overall performance, while we do not consider this aspect. Another formalism for publish/subscribe system is provided by Garlan et al. [27]. Instead of building executable programs as we do, they rely on a finite state machine that can be checked using existing model checking tools.

Chapter 3

Simple Hierarchical Variability Models

3.1 Introduction

System diversity is prevalent in modern software systems. Systems simultaneously exist in many different variants in order to comply with different requirements. Software product line engineering [54] aims at developing a family of system variants by managed reuse in order to decrease time to market and to improve quality. The variability of the different products in a software product line can be represented at different levels [18]. *Problem space variability* describes product variation in terms of features where a feature is a user-visible product characteristic. The set of valid feature configurations defines the set of possible products. However, features do not relate to the actual artifacts that are used to realize the products.

Instead, *solution space variability* captures the variability of the product line in terms of shared artifacts that are used to build the actual products. In this chapter, we capture solution space variability in terms of variable artifact implementations for fixed artifact names. This means that in different product variants an artifact with the same name can be realized with different implementations. An artifact in this context can be a component at a suitable level of granularity, such as a method, a class, or a module. Then, the artifact name would be the method signature (including the method name), interface signature or module signature, respectively, while the artifact implementation would be the method body, interface implementation, or the module realization. In [57] we used the finest of these levels, i.e., method signatures and method bodies, while in Section 3.4 we show another interpretation, where artifact names are types and artifact implementations are classes, interfaces or types implementing the former type.

In order to describe the relationship of the artifact names to the artifact implementations in the product variants, we introduce *hierarchical variability models*. Hierarchical variability models represent, in a hierarchical manner, the artifact implementations that are common to all products and the variations in the artifact implementations that can occur between different products. On each hierarchical level, there is a *common set* of artifact implementations that represent parts shared by all products, while *variation points* represent parts that can vary from product to product. Every variation point is associated with a set of variants that represent choices for realizing the variation point in different ways. A variant is itself represented by a hierarchical variability model, introducing a new level of hierarchy. A product described by a hierarchical variability model is obtained by selecting a variant at every variation point. Hierarchical variability models support modular design (as we show in [31]) and divide-and-conquer reasoning for product lines, such as the formal verification of critical requirements of all products of a family (as we illustrate in [57]).

In this chapter, we propose a hierarchical variability model that is *simple* in the sense that it requires the choice of exactly one variant for every variation point, and does not specify any constraints between choices made at different variation points. Figure 3.1 shows a simple hierarchical variability model for a cash desk system, depicted as a tree with a root node marked CashDesk. Common to all cash desk systems are the following artifact implementations: $saleProcess_{\mathsf{CashDesk}}$ for handling the sale process, and two implementations called $writeReceipt_{\mathsf{CashDesk}}$ and $updateStock_{\mathsf{CashDesk}}$ responsible for the corresponding

tasks. The notational convention is that an artifact implementation is an artifact name (*e.g.*, *saleProcess*) with an index (*e.g.*, CashDesk).

At the first level of hierarchy, a cash desk can vary in two uncorrelated (*i.e.*, orthogonal) aspects. First, there are two methods to input data about merchandise payed for at the cash desk: by keyboard or using a scanner. Second, there are two ways to pay, either in cash or by card. Thus, on the first level, the hierarchical variability model in the figure has two variation points: InputMethods and PaymentMethods. Each variation point has associated variants which capture one particular way of realizing the variation point. Variation point InputMethods has two variants, Keyboard and Scanner, each with an implementation of the corresponding input method. Variation point PaymentMethods also has two variants: Cash and Card for the two forms of payment. Both variants provide an artifact named *slot* for inserting the means of payment and *pay* for the actual payment process with different implementations. *slot* has one implementation in each variant, whereas *pay* has one implementation for cash and three variant implementations for card, corresponding to three different types of card.

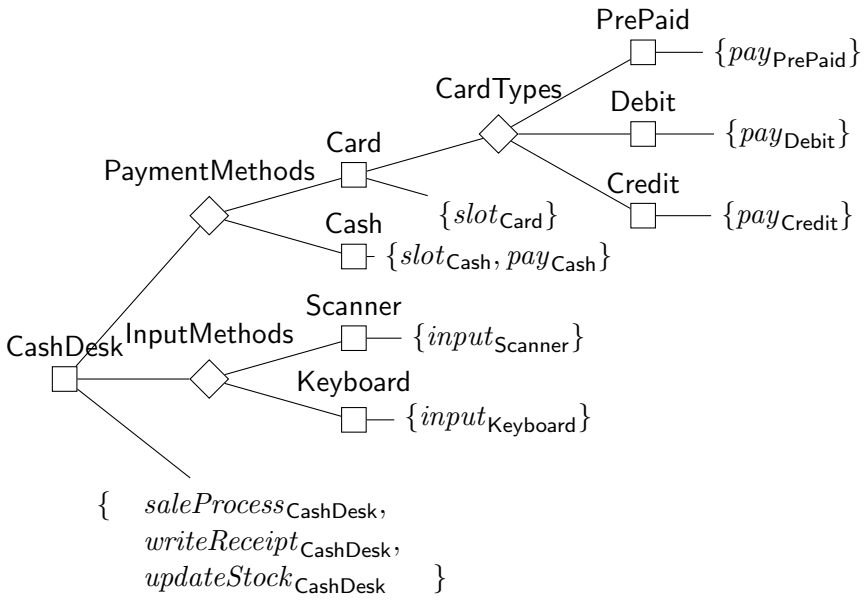

Figure 3.1: The CashDesk hierarchical variability model (drawn sideways).

Simple hierarchical variability models are in general not unique with respect to the product families they define. For instance, a trivial hierarchical variability model for the family defined by the hierarchical variability model in Figure 3.1, is the one which has no common artifact implementations, but only one variation point with variants for every product of the family. This model defines the same family, but contains the same artifact implementations several times in different variants. The intention of a hierarchical variability model is that, on each level of hierarchy, *common sets* of artifact implementations are factored out, while *uncorrelated* (or *orthogonal*) sets of artifact implementations are delegated to different variation points. To provide a measure for the quality of hierarchical variability models for defining a family in an economical way, we define the *separation degree* of a model (Definition 3.2.6) as the ratio between the total number of artifact implementations from which products are constructed and the total number of artifact implementation occurrences in the common sets of the model. Thus, high-quality models capture repetitions across products in a family without repetition in the model. The maximal possible separation degree of one is reached in models where every artifact implementation occurs in exactly one common set.

In order to reason formally about hierarchical variability models, we provide these with a formal semantics in terms of the products that can be generated by variant selection. We define *well-formedness constraints* on hierarchical variability models, under which the separation degree of the model is equal to one by construction. We term the class of families generated by well-formed variability models *simple families*,

and define this class in a model-independent fashion. We present a transformation from simple families to hierarchical variability models that (re)constructs the unique well-formed model that generates the family. Uniqueness is established by showing that the two transformations—from well-formed models to simple families and *vice versa*—are inverses of each other. For practical purposes, the latter transformation can be used for *variability model mining* from a given family of products.

To the best of our knowledge, this work is the first to provide a formal semantics for hierarchical variability models in the solution space, and to characterize a class of variability models through the class of generated product families. Previous work has been informal, as for instance the Koala component model [60], hierarchical variability modeling for software architectures [31], or plastic partial components [53]. Our work is also the first to provide a technique for constructing a hierarchical solution space variability model from a given family. Due to their conceptual simplicity, our formalism and its characterization provide a suitable starting point for the study and comparison of different product line models, such as variability models with optional or multiple variant selections, or with requires/excludes constraints between variants. For such models, uniqueness of the representation may not be guaranteed.

Our main contributions are thus:

(i) A formal definition of *simple families* as families that can be formed from artifact implementations by using a set of base operations on families (Section 3.2.1).

(ii) A definition of *simple hierarchical variability models*, together with a quality measure called *separation degree* and a set of *well-formedness constraints* yielding (by construction) models with maximal measure (Section 3.2.2).

(iii) A formal semantics for hierarchical variability models in terms of *family generation*, and a proof that, for every well-formed variability model, the generated family is simple (Section 3.3.1).

(iv) A procedure to construct hierarchical variability models from simple families that produces well-formed models (Section 3.3.2).

(v) A *characterization result* stating that, for well-formed hierarchical variability models and simple families, family generation and hierarchical variability model construction are *inverses* of each other, thus implying correctness of model construction and uniqueness of well-formed models with respect to the families they generate (Section 3.3.3).

In Section 3.4 we illustrate the application of our technique for variablity model mining on a small product line of Java classes for a simple data structure, where the automatically extracted hierarchical model allows to capture the various implemented ways of representing and outputting the data structure. We discuss related work in Section 3.5, and conclude in Section 3.6 with conclusions and directions for future work.

3.2 Families and Variability Models

In this section, we present product families as a semantic domain for our hierarchical variability model. The model is presented in the following subsection.

3.2.1 Families

We consider products realized by a set of artifact implementations for a given set of artifact names. An artifact can be thought of as, *e.g.*, a component or a method. We fix a countably infinite set of artifact names Art.

Definition 3.2.1 (Product, family)**.** *An* artifact implementation *is an indexed artifact name; let a_i denote the i-th implementation of artifact name a. A* product P *is a finite set of artifact implementations, where for each artifact name there is at most one implementation. A* family $\mathcal{F}$ *is a finite non-empty set of products.*

Thus, products can be seen as partial maps from artifact names to natural numbers, having a finite domain; we use Nat^{Art} to denote the set of all products over Art. We refer to singleton set families as *core* families, or simply cores. The family consisting of the empty product is denoted $1_{\mathcal{F}}$.

Example 6. *Here are some families that are used later to illustrate various notions.*

$$\begin{aligned}\mathcal{F}_A = \{\ &\{a_1,b_1,c_1,d_1,e_1\},\{a_1,b_1,c_1,d_1,e_2\},\{a_1,b_1,c_2,d_2,e_1\},\\ &\{a_1,b_1,c_2,d_2,e_2\},\{a_1,b_1,c_2,d_3,e_1\},\{a_1,b_1,c_2,d_3,e_2\}\ \}\\ \mathcal{F}_B = \{\ &\{a_1,b_1\},\{a_1,b_2\},\{a_2,b_1\}\ \}\end{aligned}$$

Next, we define two mappings for identifying the artifact names and artifact implementations that occur in a family.

Definition 3.2.2 (Family names and implementations)**.** *The mapping* $\text{names}(\mathcal{F})$ *from families to sets of artifact names and the mapping* $\text{impls}(\mathcal{F})$ *from families to sets of artifact implementations are defined as follows, where* $a^1,\ldots,a^n \in Art$ *and* $i_1,\ldots,i_n \in Nat$*:*

$$\begin{aligned}\text{names}(\mathcal{F}) &\stackrel{\text{def}}{=} \textstyle\bigcup_{P\in\mathcal{F}} \text{names}(P)\\ \textit{where } \text{names}(\{a^1_{i_1},\ldots,a^n_{i_n}\}) &\stackrel{\text{def}}{=} \{a^1,\ldots,a^n\}\\ \text{impls}(\mathcal{F}) &\stackrel{\text{def}}{=} \textstyle\bigcup_{P\in\mathcal{F}} \text{impls}(P)\\ \textit{where } \text{impls}(\{a^1_{i_1},\ldots,a^n_{i_n}\}) &\stackrel{\text{def}}{=} \{a^1_{i_1},\ldots,a^n_{i_n}\}\end{aligned}$$

In this definition we abuse notation by also defining mappings with the same names from products to the same co-domains.

We use two binary operations on families, the usual set union operation $\cup$ and the *product union* operation $\bowtie$ over families with disjoint sets of artifact names defined by:

$$\mathcal{F}_1 \bowtie \mathcal{F}_2 \stackrel{\text{def}}{=} \{P_1 \cup P_2 \mid P_1 \in \mathcal{F}_1 \wedge P_2 \in \mathcal{F}_2\}$$

and generalized through $\prod_{i\in I}\mathcal{F}_i$ to non-empty sets of families. Intuitively, the product union of two families is the family having as products all possible combinations of products of the original families. Both operations are commutative and associative.

We now define a distinct class of families that we later relate to a specific class of hierarchical variability models. The class of families contains all single-product families consisting of a single artifact implementation, and is closed under product union of families over disjoint sets of artifact names, and under union of families over the same set of artifact names, but having disjoint implementations.

Definition 3.2.3 (Simple family)**.** *The class* $\boldsymbol{F}$ *of* simple *families is the least set of families closed under the formation rules:*

(F1) $\{\{a_i\}\} \in \boldsymbol{F}$ *for any* $a \in Art$ *and* $i \in Nat$*.*

(F2) $\mathcal{F}_1 \bowtie \mathcal{F}_2 \in \boldsymbol{F}$ *for any* $\mathcal{F}_1,\mathcal{F}_2 \in \boldsymbol{F}$ *such that* $\text{names}(\mathcal{F}_1) \cap \text{names}(\mathcal{F}_2) = \emptyset$*.*

(F3) $\mathcal{F}_1 \cup \mathcal{F}_2 \in \boldsymbol{F}$ *for any* $\mathcal{F}_1,\mathcal{F}_2 \in \boldsymbol{F}$ *such that* $\text{names}(\mathcal{F}_1) = \text{names}(\mathcal{F}_2)$ *and* $\text{impls}(\mathcal{F}_1) \cap \text{impls}(\mathcal{F}_2) = \emptyset$*.*

Example 7. *The family* $\{\{a_1,b_1\},\{a_1,b_2\}\}$ *is simple, as it can be presented as* $\{\{a_1\}\} \bowtie (\{\{b_1\}\} \cup \{\{b_2\}\})$ *which follows the above formation rules. Family* $\mathcal{F}_A$ *of Example 6 is also simple (as we shall see later in Example 11), while family* $\mathcal{F}_B$ *of Example 6 is not: there is no way of building this family with the above formation rules.*

To characterize the applicability of the formation rules, we introduce the concept of correlation between artifact names as a restriction on the possible combinations of their implementations. If two artifact names are correlated, then not all possible combinations of artifact implementations occur in the family which means that the artifact implementations depend on each other.

More formally, two distinct artifact names $a, b \in \text{names}(\mathcal{F})$ are *correlated* in a family $\mathcal{F}$, denoted $a\, C_{\mathcal{F}}\, b$, if there are implementations $a_i, b_j \in \text{impls}(\mathcal{F})$ such that no product in $\mathcal{F}$ contains both implementations simultaneously. Otherwise, names a and b are termed *uncorrelated* or *orthogonal*. The correlation relation $C_{\mathcal{F}}$ on names $(\mathcal{F})$ is symmetric, and hence, its reflexive and transitive closure $C^*_{\mathcal{F}}$ is an equivalence relation. As usual, we denote the partitioning induced by $C^*_{\mathcal{F}}$ on names $(\mathcal{F})$ by names $(\mathcal{F})\,/C^*_{\mathcal{F}}$ (quotient set).

Example 8. *Consider family $\mathcal{F}_A$ of Example 6. The only two correlated names are c and d, evidenced by the lack of a product containing, for instance, c_1 and d_2. Thus, we have* $\text{names}(\mathcal{F}_A)\,/C^*_{\mathcal{F}_A} = \{\{a\}, \{b\}, \{c, d\}, \{e\}\}$.

Correlation (and orthogonality) extends naturally to products in a family: Products P and P' are correlated in $\mathcal{F}$ if some artifact name occurring in P is correlated to some artifact name occurring in P'. Similarly, we define the sharing relation $N_{\mathcal{F}}$ on $\mathcal{F}$ as $P_1\, N_{\mathcal{F}}\, P_2 \overset{\text{def}}{\Leftrightarrow} P_1 \cap P_2 \neq \emptyset$, and use its reflexive and transitive closure $N^*_{\mathcal{F}}$ to partition the family $\mathcal{F}$.

The following result provides sufficient conditions for the applicability of the three formation rules for simple families. The proof of this proposition, as all other proofs can be found in a technical report [22]. As usual, $\overline{A}$ denotes the complement of set A.

Proposition 1. *Let family $\mathcal{F}$ be simple. The following holds.*

(i) Let $a_i \in \text{impls}(\mathcal{F})$, and let $\mathcal{F}'$ be the projection of $\mathcal{F}$ on $\text{names}(\mathcal{F}) \setminus \{a\}$. *$a_i$ occurs in all products of $\mathcal{F}$,* i.e., *$a_i \in \bigcap_{P \in \mathcal{F}} P$, iff $\mathcal{F} = \{\{a_i\}\} \bowtie \mathcal{F}'$. Then either $\mathcal{F}' = 1_{\mathcal{F}}$ and thus rule (F1) applies, or else $\mathcal{F}'$ is simple and rule (F2) applies.*

(ii) Let $\{A_1, A_2\}$ be a non-trivial partitioning of $\text{names}(\mathcal{F})$, *and let $\mathcal{F}_1$ and $\mathcal{F}_2$ be the projections of $\mathcal{F}$ on A_1 and A_2, respectively. Every name in A_1 is orthogonal to every name in A_2 in $\mathcal{F}$,* i.e., *$A_1 \times A_2 \subseteq \overline{C_{\mathcal{F}}}$, iff $\mathcal{F} = \mathcal{F}_1 \bowtie \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule (F2) applies in this case.*

(iii) Let $\{\mathcal{F}_1, \mathcal{F}_2\}$ be a non-trivial partitioning of $\mathcal{F}$. No product of $\mathcal{F}_1$ shares an artifact implementation with any product of $\mathcal{F}_2$, i.e., *$\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \overline{N_{\mathcal{F}}}$, iff $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ and $\mathcal{F}_1$ and $\mathcal{F}_2$ are simple. Formation rule (F3) applies in this case.*

The following important property of simple families follows from the above result: If a simple family $\mathcal{F}$ can be formed by formation rule (F2) with some suitable $\mathcal{F}_1$ and $\mathcal{F}_2$ satisfying the rule's condition, then it cannot be formed by formation rule (F3), and *vice versa*. Thus, when restricted to simple families, the two operations on families do not distribute over each other. This entails that simple families have *unique* formation trees modulo commutativity and associativity of the two operations associated with the rules.

3.2.2 Variability Models

In order to represent solution space variability of families in terms of shared artifact implementations, we consider simple hierarchical variability models.

Definition 3.2.4 (Simple hierarchical variability model)**.** *A* simple hierarchical variability model (SHVM) *$\mathcal{S}$ is inductively defined as:*

(i) a (possibly empty) common set *of artifact implementations M_C, or*

(ii) a pair $(M_C, \{VP_1, \ldots, VP_n\})$ where M_C is defined as above and the set $\{VP_1, \ldots, VP_n\}$ of variation points *is non-empty. A variation point $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$, where $k_i \geq 2$, is a set of (at least two) SHVMs called* variants.

We sometimes refer to an SHVM simply as a variability model. An SHVM with only a common set of artifact implementations is called *ground model*. An SHVM generates a family $\mathcal{F}$ through all possible ways of resolving the variabilities of the SHVM. This process recursively selects exactly one variant for each variation point. We defer a formal definition of such a semantics for SHVMs to Section 3.3.1. Variability models can be naturally depicted as trees, where leaves are common sets of artifact implementations, and internal nodes are the roots of SHVMs or variation points.

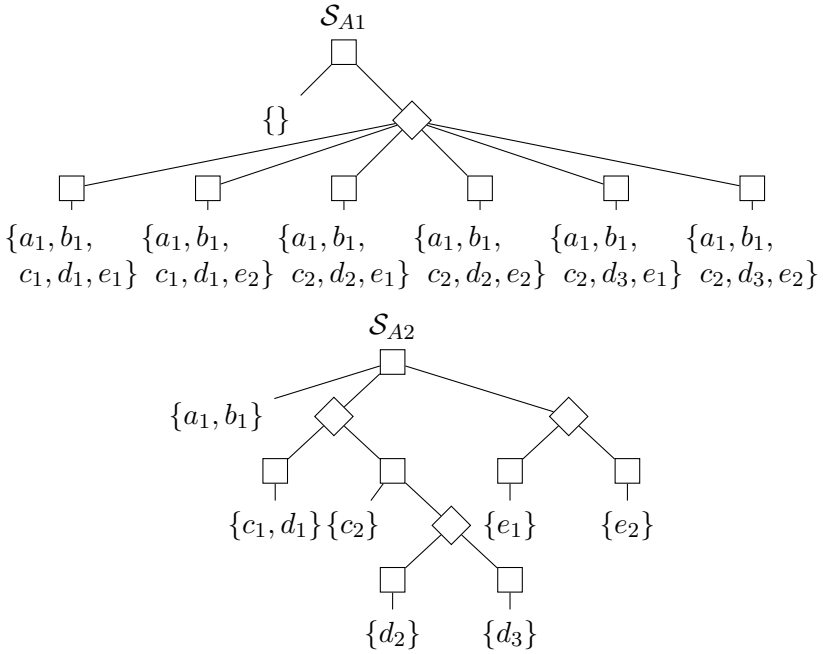

Figure 3.2: SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ for the family $\mathcal{F}_A$ in Example 6.

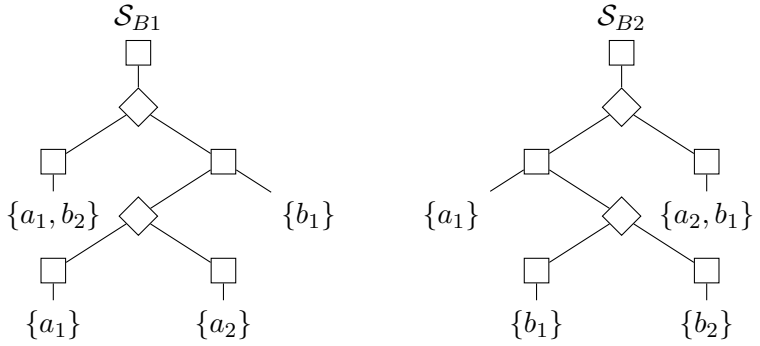

Figure 3.3: SHVMs $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ for the family $\mathcal{F}_B$ in Example 6.

Example 9. *Figure 3.2 and Figure 3.3 show four variability models named $\mathcal{S}_{A1}$, $\mathcal{S}_{A2}$, $\mathcal{S}_{B1}$, and $\mathcal{S}_{B2}$. In these figures, (sub)trees showing variability models are rooted with boxes, and subtrees showing variation points are rooted with diamonds.*

In analogy with Definition 3.2.2, we define two mappings for identifying the artifact names and artifact implementations that occur in SHVMs.

Definition 3.2.5 (SHVM names and implementations)**.** *The mapping* $\text{names}(\mathcal{S})$ *from SHVMs to sets of artifact names and the mapping* $\text{impls}(\mathcal{S})$ *from SHVMs to sets of artifact implementations are defined as*

follows, where $a^1, \ldots, a^n \in Art$ *and* $i_1, \ldots, i_n \in Nat$*:*

$$\begin{aligned}
\text{names}\left(\{a^1_{i_1}, \ldots, a^n_{i_n}\}\right) &\stackrel{\text{def}}{=} \{a^1, \ldots, a^n\} \\
\text{names}\left((M_c, \{VP_1, \ldots, VP_n\})\right) &\stackrel{\text{def}}{=} \text{names}\left(M_C\right) \cup \textstyle\bigcup_{1 \le i \le n} \text{names}\left(VP_i\right) \\
\textit{where } \text{names}\left(VP\right) &\stackrel{\text{def}}{=} \textstyle\bigcup_{\mathcal{S} \in VP} \text{names}\left(\mathcal{S}\right) \\
\text{impls}(\{a^1_{i_1}, \ldots, a^n_{i_n}\}) &\stackrel{\text{def}}{=} \{a^1_{i_1}, \ldots, a^n_{i_n}\} \\
\text{impls}((M_c, \{VP_1, \ldots, VP_n\})) &\stackrel{\text{def}}{=} \text{impls}(M_C) \cup \textstyle\bigcup_{1 \le i \le n} \text{impls}(VP_i) \\
\textit{where } \text{impls}(VP) &\stackrel{\text{def}}{=} \textstyle\bigcup_{\mathcal{S} \in VP} \text{impls}(\mathcal{S})
\end{aligned}$$

Again we abuse notation by also defining mappings with the same names from variation points to the same co-domains.

Next, we define a measure of the degree of separation in a variability model, as the proportion between the number of artifact implementations of a variability model and the total size of the leaves of the SHVM tree. The separation degree is, thus, a number in the interval $\langle 0, 1]$, and captures the degree to which the commonalities and orthogonalities of products are factored out as common sets and variation points in a variability model, respectively: the higher this degree, the less artifact implementations occur repeatedly in more than one leaf. The maximal value of 1 holds when every artifact implementation occurs in exactly one leaf; this is trivially the case for ground models.

Definition 3.2.6 (Separation degree)**.** *The* separation degree $sd(\mathcal{S})$ *of a variability model* $\mathcal{S}$ *is defined as:*

$$\begin{aligned}
sd(\{\}) &\stackrel{\text{def}}{=} 1 \\
sd(\mathcal{S}) &\stackrel{\text{def}}{=} \frac{|\text{impls}(\mathcal{S})|}{sd'(\mathcal{S})} \qquad \textit{if } \mathcal{S} \neq \{\}
\end{aligned}$$

where $sd'(\mathcal{S})$ *is inductively defined as follows:*

$$\begin{aligned}
sd'(M_C) &\stackrel{\text{def}}{=} |M_C| \\
sd'((M_C, \{VP_1, \ldots, VP_n\})) &\stackrel{\text{def}}{=} sd'(M_C) + \Sigma_{1 \le i \le n} sd'(VP_i) \\
\textit{where } sd'(VP) &\stackrel{\text{def}}{=} \Sigma_{\mathcal{S} \in VP} sd'(\mathcal{S})
\end{aligned}$$

As usual $|S|$ denotes the cardinality of set S.

The following definition provides a set of well-formedness constraints on SHVMs. The separation degree of variability models satisfying these constraints is always one as we show in Proposition 2.

Definition 3.2.7 (Well-formed variability model)**.** *A ground variability model* $\mathcal{S} = M_C$ *is* well-formed *if constraint (S1) below is satisfied. A variability model* $\mathcal{S} = (M_C, \{VP_1, \ldots, VP_n\})$ *with variation points* $VP_i = \{\mathcal{S}_{i,j} \mid 1 \le j \le k_i\}$ *is* well-formed *if all variants* $\mathcal{S}_{i,j}$ *are well-formed, and furthermore, the following constraints are satisfied:*

(S1) M_C *implements artifact names at most once.*

(S2) $\text{names}\left(M_C\right) \cap \text{names}\left(VP_i\right) = \emptyset$ *for all* i*, and*
$\text{names}\left(VP_{i_1}\right) \cap \text{names}\left(VP_{i_2}\right) = \emptyset$ *whenever* $i_1 \neq i_2$*.*

(S3) $\text{names}\left(\mathcal{S}_{i,j_1}\right) = \text{names}\left(\mathcal{S}_{i,j_2}\right)$ *for all* i, j_1, j_2*, and*
$\text{impls}(\mathcal{S}_{i,j_1}) \cap \text{impls}(\mathcal{S}_{i,j_2}) = \emptyset$ *whenever* $j_1 \neq j_2$*.*

Example 10. *Consider the SHVMs* $\mathcal{S}_{A1}$ *and* $\mathcal{S}_{A2}$ *depicted in Figure 3.2.* $\mathcal{S}_{A1}$ *is not well-formed whereas* $\mathcal{S}_{A2}$ *is. The separation degrees are* $sd(\mathcal{S}_{A1}) = \frac{9}{6 \cdot 5} = 0.3$ *and* $sd(\mathcal{S}_{A2}) = \frac{9}{9} = 1$*. Figure 3.3 depicts another two SHVMs,* $\mathcal{S}_{B1}$ *and* $\mathcal{S}_{B2}$*. Neither of these are well-formed and both have separation degree* $\frac{4}{5} = 0.8$*.*

The constraints in Definition 3.2.6 ensure that the separation degree of a well-formed SHVM is equal to one, and is thus maximal.

Proposition 2. *If variability model $\mathcal{S}$ is well-formed then $sd(\mathcal{S}) = 1$.*

Note that the converse of Proposition 2 does not hold in general: The variability model $M_C = \{a_1, a_2\}$ has separation degree 1, but well-formedness constraint $(\mathcal{S}1)$ is not satisfied.

3.3 Relating Families and Variability Models

In this section, we present translations between well-formed variability models and simple families, and show that they are inverses of each other. In particular, this entails that the translation from simple families to variability models produces the unique well-formed model generating the respective family, thus giving a procedure for constructing a variability model from a given family.

3.3.1 From Variability Models to Families

The set of products generated by a ground model is the singleton set comprising the set of common artifact implementations (and, thus, representing one product). The set of products generated by a variation point is the union of the product sets generated by its variants. Finally, the set of products generated by an SHVM with a non-empty set of variation points is the set of all products consisting of the common artifact implementations and of exactly one product from the set generated by each variation point.

Definition 3.3.1 (Family generation)**.** *The mapping $family(\mathcal{S})$ from variability models to families is inductively defined as follows:*

$$family(M_C) \stackrel{\text{def}}{=} \{M_C\}$$
$$family((M_C, \{VP_1, \ldots, VP_n\})) \stackrel{\text{def}}{=} \{M_C\} \bowtie \prod_{1 \leq i \leq n} family(VP_i)$$
$$\text{where } family(VP) \stackrel{\text{def}}{=} \bigcup_{\mathcal{S} \in VP} family(\mathcal{S})$$

We say that variability model $\mathcal{S}$ generates *family $family(\mathcal{S})$.*

Here we again abuse notation by also defining a mapping with the same name from variation points to the same co-domain. Family generation is well-defined in the sense that well-formed variability models generate simple families.

Proposition 3. *If variability model $\mathcal{S}$ is well-formed, then $family(\mathcal{S})$ is simple.*

Example 11. *SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ in Figure 3.2 both generate family $\mathcal{F}_A$ in Example 6, implying that family $\mathcal{F}_A$ is simple since $\mathcal{S}_{A2}$ is well-formed. SHVMs $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ in Figure 3.2 both generate family $\mathcal{F}_B$ in Example 6. Of these four, $\mathcal{S}_{A2}$, $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ have maximal separation degree in the sense that, for each of the families $\mathcal{F}_A$ and $\mathcal{F}_B$, no other SHVMs for the same family have higher separation degree.*

3.3.2 From Families to Variability Models

We now present a reverse transformation from simple families to well-formed variability models. Recall that simple families have unique formation trees modulo commutativity and associativity of the two operations. Well-formed SHVMs can thus be seen as a uniform way of grouping the formation terms. Every family $\mathcal{F}$ can be decomposed into the form:

$$\mathcal{F} = \{P\} \bowtie \mathcal{F}_V, \quad \mathcal{F}_V = \prod_{1 \leq i \leq n} \mathcal{F}_i, \quad \mathcal{F}_i = \bigcup_{1 \leq j \leq k_i} \mathcal{F}_{i,j}$$

where P is a product, or equivalently, as a single equation:

$$\mathcal{F} = \{P\} \bowtie \prod_{1 \le i \le n} \bigcup_{1 \le j \le k_i} \mathcal{F}_{i,j} \qquad (*)$$

The existence of the decomposition is ensured since every family $\mathcal{F}$ can be trivially decomposed as $\{\emptyset\} \bowtie \prod \bigcup \mathcal{F}$, *i.e.*, with product P being empty and $n = k_1 = 1$. Decomposition $(*)$ is only unique under additional constraints, under which the decomposition is called canonical.

Definition 3.3.2 (Canonical form)**.** *A family $\mathcal{F}$, decomposed as equation $(*)$ above, is in* canonical form *if the following conditions hold:*

(C1) The product P is the set of artifact implementations that are common to all products in $\mathcal{F}$.

(C2) The set of artifact names in $\mathcal{F}_V$ has n equivalence classes w.r.t. correlated artifact names $C^*_{\mathcal{F}_V}$, and for the i-th equivalence class, the family $\mathcal{F}_i$ is the projection of $\mathcal{F}_V$ onto the artifact names of the class.

(C3) For all i, $1 \le i \le n$, $\mathcal{F}_{i,j}$ are the k_i equivalence classes of $\mathcal{F}_i$ w.r.t. implementation sharing $N^*_{\mathcal{F}_i}$.

A consequence of the following proposition is that definitions and proofs may exploit the canonical form to proceed by induction on the size of simple families.

Proposition 4. *If $\mathcal{F}$ is a simple non-core family in canonical form then for all i, $1 \le i \le n$, $k_i \ge 2$ and all $\mathcal{F}_{i,j}$ are simple and of strictly smaller size than $\mathcal{F}$.*

The decomposition into canonical form is clearly unique for a simple family, and exposes one level of hierarchy. Thus, by iterative application of the decomposition, we obtain a mapping from families to hierarchical variability models.

Definition 3.3.3 (Variability model generation)**.** *The mapping $shvm(\mathcal{F})$ from simple families presented in canonical form to variability models is inductively defined as follows:*

$$shvm(\{M_C\}) \stackrel{\text{def}}{=} M_C$$

$$shvm\big(\{M_C\} \bowtie \textstyle\prod_{1 \le i \le n} \bigcup_{1 \le j \le k_i} \mathcal{F}_{i,j}\big) \stackrel{\text{def}}{=} (M_C, \{VP_1, \ldots, VP_n\})$$

$$\text{where } VP_i \stackrel{\text{def}}{=} \{shvm(\mathcal{F}_{i,j}) \mid 1 \le j \le k_i\}$$

We say that family $\mathcal{F}$ generates *variability model $shvm(\mathcal{F})$.*

Proposition 4 guarantees that the above definition is well-defined, in the sense that $shvm(\mathcal{F})$ is indeed an SHVM. Furthermore, as the next result shows, the variability model is well-formed.

Proposition 5. *If family $\mathcal{F}$ is simple, then $shvm(\mathcal{F})$ is well-formed.*

Example 12. *Consider the family $\mathcal{F}_A$ from Example 6.*

- *In the first step of the decomposition of $\mathcal{F}_A$ into canonical form we obtain the common set $P = \{a_1, b_1\}$ and the family $\mathcal{F}_V = \{\{c_1, d_1, e_1\}, \{c_1, d_1, e_2\}, \{c_2, d_2, e_1\}, \{c_2, d_2, e_2\}, \{c_2, d_3, e_1\}, \{c_2, d_3, e_2\}\}$.*
- *In the next step, we analyze $\mathcal{F}_V$ to find that only artifact names c and d are correlated. Projecting $\mathcal{F}_V$ onto the two resulting equivalence classes $\{c, d\}$ and $\{e\}$ we obtain the two variation points $\mathcal{F}_1 = \{\{c_1, d_1\}, \{c_2, d_2\}, \{c_2, d_3\}\}$ and $\mathcal{F}_2 = \{\{e_1\}, \{e_2\}\}$.*
- *In the third step, we analyze $\mathcal{F}_1$ and see that two products share the artifact implementation c_2, which gives us the variants $\mathcal{F}_{1,1} = \{\{c_1, d_1\}\}$ and $\mathcal{F}_{1,2} = \{\{c_2, d_2\}, \{c_2, d_3\}\}$, and then analyze $\mathcal{F}_2$ to obtain the variants $\mathcal{F}_{2,1} = \{\{e_1\}\}$ and $\mathcal{F}_{2,2} = \{\{e_2\}\}$.*

Only $\mathcal{F}_{1,2}$ is not a ground model. Applying the above steps decomposes it into a common set $\{c_2\}$ and a single variation point with two variants consisting of the common sets $\{d_2\}$ and $\{d_3\}$. It is easy to see that $shvm(\mathcal{F}_A)$ is the variability model $\mathcal{S}_{A2}$ in Figure 3.2.

3.3.3 Characterization Results

Our first result establishes correctness of model extraction.

Lemma 1. *For every simple family $\mathcal{F}$ we have:*

$$family(shvm(\mathcal{F})) = \mathcal{F}$$

The second result establishes uniqueness of well-formed models w.r.t. the generated (simple) family.

Lemma 2. *For every well-formed variability model $\mathcal{S}$ we have:*

$$shvm(family(\mathcal{S})) = \mathcal{S}$$

An immediate consequence of the above two lemmata is our main characterization result, which essentially states that the two transformations relating variability models and families are inverses of each other.

Theorem 3.3.4 (Characterization Theorem)**.** *For every simple family $\mathcal{F}$ and every well-formed variability model $\mathcal{S}$ we have:*

$$family(\mathcal{S}) = \mathcal{F} \iff shvm(\mathcal{F}) = \mathcal{S}$$

3.4 Application

In this section, we show how to apply our theory to families consisting of products of program code. We explain how to obtain an SHVM from a set of products, and what insights one can gain from the derived model. Our running example (Section 3.4.1) is a simple product family written in Java, but the application of our theory is not restricted to particular programming languages or paradigms. For example, obtaining an SHVM from a set of products written in ABS could be done in a similar manner to that shown in this section.

3.4.1 Example Product Line: Storing and Processing Collections

The example family consists of six products, where each product is a Java class. The code for all products appears in Figure 3.4.[1] The six products—named $P_{\texttt{X1}}$, $P_{\texttt{X2}}$, $P_{\texttt{Y1}}$, $P_{\texttt{Y2}}$, $P_{\texttt{Z1}}$, and $P_{\texttt{Z2}}$ after the respective class—have the following commonalities: They all store a collection of values of the custom type `Elem`, have a method for setting this state to some value, a method `process()`, and last a method `compute()` which returns some subclass of `Number`. The products have the following differences: The type of the state is either `List` or `Set`, both subinterfaces of `java.util.Collection`. In the case of `List`, method `compute()` returns a `Double`, and in the case of `Set`, it returns either a `Byte` or an `Integer`. Furthermore, method `process()` either prints out the state one element at a time using a method on class `System`, or it produces a `String` from the elements and returns it.

3.4.2 From Code to Artifacts

Before we can construct an SHVM, we need a scheme to obtain a set of products, that is, products in the sense of Definition 3.2.1. Thus, we must identify artifacts in the product code. An artifact name in the program code is a construct that may occur several times, but with different realizations when there are several artifact implementations. Deciding how to identify artifacts in the code means determining what are the important parts of the code for the variability model of the product line. In general, this can be done in many ways. Here, we give one possible example.

For this example, we consider an artifact to be a pair of Java types, one being the name of the type and one being its implementation. For two Java types to form an artifact, they must be connected as

```
1  class X1 {
       List<Elem> state = new ArrayList<Elem>();
3
       void setState(List<Elem> arg) {
5          this.state.addAll(arg);
       }
7
       Double compute() { ... }
9
       void process() {
11         for (Elem e : state)
               System.out.println(e);
13 }   }
   class X2 {
15     List<Elem> state = new ArrayList<Elem>();

17     void setState(List<Elem> arg) { ... } // as before

19     Double compute() { ... }

21     String process() {
           String res = "";
23         for (Elem e : state)
               res = res + "," + e.toString();
25         return res;
   }   }
27 class Y1 {
       Set<Elem> state = new HashSet<Elem>();
29
       void setState(Set<Elem> arg) {
31         this.state.addAll(arg);
       }
33
       Byte compute() { ... }
35
       void process() { ... } // as before with same sig.
37 }
   class Y2 {
39     Set<Elem> state = new HashSet<Elem>();

41     void setState(Set<Elem> arg) { ... } // as before

43     Byte compute() { ... }

45     String process() { ... } // as before with same sig.
   }
47 class Z1 {
       Set<Elem> state = new HashSet<Elem>();
49
       void setState(Set<Elem> arg) { ... } // as before
51
       Integer compute() { ... }
53
       void process() { ... } // as before with same sig.
55 }
   class Z2 {
57     Set<Elem> state = new HashSet<Elem>();

59     void setState(Set<Elem> arg) { ... } // as before

61     Integer compute() { ... }

63     String process() { ... } // as before with same sig.
   }
```

Figure 3.4: Example product line consisting of six Java classes.

Art. name	*Art. impl.*	*Connection*	*Notation*
interface I	class C	C implements I	I_C
interface I	interface J	J subinterface of I	I_J
class C	class D	D subclass of C	C_D
type T	type T	(by convention)	T_T

Table 3.1: Scheme for obtaining artifacts from Java code.

shown in Table 3.1. The types that form artifacts in our example are underlined in Figure 3.4. (Class `java.lang.Object` and interface `Collection` do not occur in the figure, but are also used.) These are some artifacts identified in the example:

- Interface `java.util.List` is connected to interface `java.util.Collection` via the Java implements relation, giving rise to the artifact $\texttt{Collection}_{\texttt{List}}$ (omitting package prefixes).
- Class `java.lang.String` is connected to the class `java.lang.Object` via the subclass relation, so we have the artifact $\texttt{Object}_{\texttt{String}}$.
- Class `Elem` is—by the convention—related to itself, so we have the artifact $\texttt{Elem}_{\texttt{Elem}}$.

With the scheme in Table 3.1, we identify the following set of products which is a simple family and yields a hierarchical variability model with three variation points including one inside the other.
$P_{\texttt{X1}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{List}}, \texttt{Number}_{\texttt{Double}}, \texttt{Object}_{\texttt{System}}\right\}$
$P_{\texttt{X2}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{List}}, \texttt{Number}_{\texttt{Double}}, \texttt{Object}_{\texttt{String}}\right\}$
$P_{\texttt{Y1}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{Set}}, \texttt{Number}_{\texttt{Byte}}, \texttt{Object}_{\texttt{System}}\right\}$
$P_{\texttt{Y2}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{Set}}, \texttt{Number}_{\texttt{Byte}}, \texttt{Object}_{\texttt{String}}\right\}$
$P_{\texttt{Z1}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{Set}}, \texttt{Number}_{\texttt{Integer}}, \texttt{Object}_{\texttt{System}}\right\}$
$P_{\texttt{Z2}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{Set}}, \texttt{Number}_{\texttt{Integer}}, \texttt{Object}_{\texttt{String}}\right\}$ In general, such products may not yield a simple family, and in such cases we cannot obtain an SHVM.

3.4.3 Constructing and Interpreting the SHVM

From the set of products obtained in the previous section, constructing an SHVM is straightforward by the procedure specified in Definition 3.3.3. We obtain the SHVM depicted in Figure 3.5. The SHVM in this figure is nearly identical to $\mathcal{S}_{A2}$ in Figure 3.2—differing only in the cardinality of set at the leftmost branch from the root. Hence, the construction proceeds similarly to that of Example 12. Since the family is simple, the obtained model is well-formed and, thus, optimal w.r.t. the separation degree.

The constructed SHVM may be read as a graphical summary of the textual product line description given in Section 3.4.1, focusing on Java types. Note, in particular, that the choice between `List` and `Set` is clearly visible as a variation point, and that, for example, the combination of `List` and `Byte` is not allowed by the SHVM, whereas `List` and `Double` is allowed.

3.5 Related Work

The existing approaches to represent solution space product line variability can be divided into three directions [61]. First, annotative approaches consider one model representing all products of a product line. Variant annotations, *e.g.*, using UML stereotypes [63, 29], presence conditions [17], or separate variability representations, such as orthogonal variability models [54], define which parts of the model have to be

[1] We have omitted the following: import declarations, definition of custom type `Elem`, and repeated or irrelevant code.

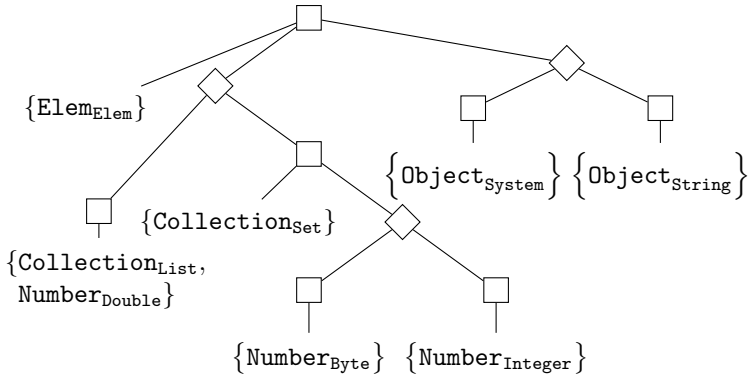

Figure 3.5: SHVM for the example family.

removed to generate the model of a concrete product. Second, compositional approaches [13, 61, 48, 9] associate product fragments with product features which are composed for particular feature configurations. Third, transformational approaches [35, 15] represent variability by rules determining how a base model has to be changed for a particular product model. All these approaches consider a representation of artifact variability without any hierarchy.

Our hierarchical variability model generalizes the ideas of the Koala component model [60] for the implementation of variant-rich component-based systems. In Koala, the variability of a component is described by the variability of its sub-components which can be selected by *switches* and explicit *diversity interfaces*. Diversity interfaces and switches in Koala can be understood as concrete language constructs targeted at the implementation level to express variation points and associated variants. Plastic partial components [53] are an architectural modeling approach where component variability is defined by extending partially defined components with variation points and associated variants. However, variants cannot contain variable components so this modeling approach is not truly hierarchical. Hierarchical variability modeling for software architectures [31] applies the modeling concepts for solution space variability presented in this chapter to component-based software engineering and provides a concrete modeling language for variable software architectures that is truly hierarchical. However, none of these approaches formally defines the semantics of hierarchical variability models, nor reasons about their well-formedness or uniqueness.

To the best of our knowledge, this chapter presents the first approach for constructing a hierarchical variability model for solution space variability from a given product family. So far, there have only been approaches to construct feature models for representing problem space variability for a given set of products. Czarnecki *et al.* [19] re-construct a feature model from a set of sample feature combinations using data mining techniques [2]. Other approaches aim at constructing feature models from sample mappings between products and their features using formal concept analysis [26], for instance, to derive logical dependencies between code variants from pre-processor annotations [58], or to construct a feature model for function-block based systems after determining model variants by similarity [56]. Loesch and Ploedereder [41] use formal concept analysis to optimize feature models in case of product line evolution, *e.g.*, to remove unused features or to combine features that always occur together. Niu and Easterbrook [47] apply formal concept analysis to functional and non-functional product line requirements in order to construct a feature model as a more abstract representation of the requirements. Also, information retrieval techniques are applied to obtain a feature model from heterogeneous product line requirements [6]. Using hierarchical clustering, a tree structure of textually similar requirements is constructed. Requirement clusters in the leaves are more similar to each other than requirements clusters closer to the root giving rise to the structure of a feature model.

In our work, we abstract from the need to determine the different variants of the same conceptual entity by assuming fixed artifact names and corresponding artifact implementations. However, if we relax this assumption, techniques, such as similarity analysis [56] or formal concept analysis [26] could be applied to infer the relationship between different variants of the same conceptual entity, and thus make our approach

applicable.

3.6 Remarks

We present hierarchical solution space variability models for software product lines. We give a formal semantics of such models in terms of sets (or families) of products, where each product is a set of artifact implementations. We introduce separation degree as a quality measure of hierarchical variability models. Well-formed variability models are identified as a class of models for which the measure is maximal (and equal to one) and which are unique for the family they generate; the class of families generated by such models is the class of simple families. We present a transformation that constructs, from a simple family, the unique well-formed model that generates it, and prove uniqueness by showing that family generation and model construction are inverses of each other for this class of models.

While maximal separation degree and uniqueness of models with maximal measure are theoretically appealing, in practice, product families might not be simple. Still, separation degree is a useful measure for hierarchical variability models, and, as Examples 10 and 11 suggest, searching for the set of models with a maximal measure (not necessarily equal to one) for a given family is equally meaningful.

Chapter 4

Symbolic Inference of Automata Models

4.1 HATS Context and Motivation

The HATS ABS language is a high-level modeling language that can be used for model-based development of software product families and evolving software systems. However, there are very few high-level modeling languages where efficient software code can be automatically generated from a model. The ABS language is no exception to this. Despite the existence of an ABS execution environment, it is doubtful whether this could be used for production quality code.

Therefore for ABS, as for other modeling languages such as UML, a gap usually emerges during development between high-level models and manually constructed low-level code. This gap manifests as discrepancies between the model behavior and the code behavior. It can be the source of many final product quality problems, including functional incorrectness. (See for example the discussion in [59]).

Despite this synchronisation problem between models and code, high-level models have been found to be an excellent source of software test cases, hence the recent emergence of model-based testing. (See for example the survey [24].) This, and the other many advantages of model-based development cannot be ignored. For this reason, the HATS research group at KTH has for some years been interested in the question of software testing, especially within model-based development frameworks such as ABS.

Machine learning has been proposed to overcome the problem of synchronisation failure between models and software, by automatically 'model-mining' high-level models from execution of low-level software code. Such inferred models can for example, be checked for consistency against the documented model. In the case of inconsistency they can either be used to highlight coding errors or to update the current model. Promising results have been obtained here, for example in the context of state transition models, by using active automata learning technology. (See for example research results of the EU FET project CONNECT [14]) There is also a high potential to exploit other machine learning techniques such as symbolic learning and inductive inference.

Within the HATS project WP3.2, the research group of Professor Karl Meinke at KTH-CSC has carried out research into *computational learning techniques to dynamically infer automata models* (including Kripke structures and Mealy machines) based on dynamic black-box observation of system behaviors. The results of this research activity in WP3.2 have been successfully applied elsewhere in HATS WP2.3 to the problem of black-box automated test case generation. In this report section, we will focus more on reporting the model-inference algorithms themselves.

One important feature of the ABS modeling language is the use of abstract data types (ADTs) as a general data modeling formalism that can be applied to many kinds of systems. While the literature on ADTs for systems modeling, verification and testing is extensive, little is known about the problem of inferring computational models (such as automata) over such systems. In fact, conventional algorithms for DFA inference are usually hard-wired for the Boolean data type. The extension to abstract data types (finite or infinite) has not been considered. The problem is non-trivial, since the finite tabular techniques used by DFA learning algorithms are unable to cope with infinite data types. Some new techniques are necessary

here. Since ADTs are an essential element of many high-level systems modeling languages including the ABS language, our research task was to find efficient symbolic learning algorithms for automata models, parameterised by a finite or infinite ADTs.

For this purpose we devised an extended Mealy automaton (EMA) model which integrates an algebraic ADT specification with an orthogonal algebraic model of a Mealy machine. As we have shown, the resulting many-sorted algebraic structure can be learned by inferring a finitely generated congruence as a confluent and terminating *string rewriting system* SRS. This representation of an EMA turns out to be very useful if we wish to model check the inferred system, in which case we can use term rewriting techniques directly on this representation.

Our main deliverable to WP3.2 is a new algebraic technique for learning EMA that we have termed *congruence generator extension* (CGE). The first CGE algorithm was published in 2010 as [42]. We are currently developing a much more efficient version of this algorithm, which is able to learn quite large automata models in a reasonable time. The new CGE algorithm is competitive with state of the art automata learning algorithms (for Boolean data types), but is much more flexible in terms of application to learning high-level models with high-level data types.

In subsection 4.1.1, we will contrast our algebraic approach to automata learning with conventional learning algorithms which use state partition tables. In subsection 4.2, we compare our approach with related computational learning approaches in the literature. In subsection 4.3, we will discuss string rewriting systems as an efficient method of representing congruences for automata models. In subsection 4.3.1. we present the CGE learning algorithm, and discuss its correctness. Finally, in subsection 4.3.2, we consider an example of CGE learning.

4.1.1 Learning Automata using Algebraic Techniques

Classical algorithms for learning a finite automaton A such as the L* algorithm of [8] typically approach the problem of learning the state space structure of A as a partition refinement problem. In this case the objects to be partitioned are a set of input strings for A, which act as synonyms for an unknown set of state names. The problem is then to determine which input strings denote the same states in A. From this information, the state transition structure of A can be inferred. Initially, one starts from the coarsest possible partition, which is the unit partition, in which all input strings are identified. To separate two strings $\overline{i}$ and $\overline{j}$ into distinct partition classes, it is necessary to find a suffix $\overline{k}$ such that the two strings $\overline{i}\,\overline{k}$ and $\overline{j}\,\overline{k}$ generate different observed behaviours from A. Different learning algorithms have different methods for splitting partition classes. However, all such methods come down to identifying inequalities $\overline{i} \neq \overline{j}$ between input strings.

On the other hand, taking a more algebraic viewpoint of the whole problem, one can consider the dual of the above approach. Starting from the finest possible partition, which is the equality relation $=$ on input strings, one can try to identify a finite set E of equations between input strings (of the form $\overline{i} = \overline{j}$), which characterise A. One can then consider the smallest congruence $\equiv^E$ generated by E. This is tantamount to learning loops and merges within the graph structure of A. The general theory of universal algebra (see for example [46]) dictates that the structure on which this finitely generated congruence acts should be an *initial algebra* I in an appropriate category of automata. Recall that an algebra I is initial (or absolutely free) in a category K, if $I \in K$ and for each $B \in K$ there exists a unique homomorphism $\beta : I \to B$. In this case B can be uniquely constructed (up to isomorphism) as the quotient algebra $B \cong I/\equiv^\beta$, where $\equiv^\beta$ is the kernel congruence of β. In particular

$$A \cong I/\equiv^E,$$

and every automaton has a unique construction in this way.

On the one hand this algebraic approach is mathematically elegant. We can abstract away a complex algorithmic construction to a much simpler algebraic construction. At the same time, we obtain a framework for studying different heuristics for learning, as the choice of different congruences within the lattice of congruences on I. On the other hand, this approach opens up new approaches to learning more complex kinds of automata models over abstract data types (ADTs), and even infinite data types and infinite state

automata. For the latter, notice that even if A and $\equiv^E$ are infinite, the generating set E could be finite and hence learnable in principle. This is clearly never the case if one tries to learn the dual set of inequalities $\overline{i} \neq \overline{j}$, which must be infinite.

In subsection 4.3.1 we present a specific learning algorithm for Mealy automata, which has been derived using the algebraic ideas sketched above. Our algorithm is termed CGE (Congruence Generator Extension) since it tries to extend the empty set of generators in a consistent but greedy way to a maximal congruence, given an underlying set Λ of observations about A. The intended application for the CGE learning algorithm is in software engineering. This application is described in detail in [43]. In this context, a number of technical features such as *incremental learning* and an absence of *membership queries* are necessary. These features also distinguish CGE from related learning algorithms in the literature.

4.2 Learning Algorithms for Automata: a Short Literature Survey.

The literature on learning algorithms for automata is extensive. Recent surveys can be found in [11], [51] and [20]. An important distinction can be made between: (i) *complete*, and (ii) *incremental learning*. For complete learning, the goal is to construct a complete and behaviourally equivalent representation of an unknown automaton A which is called the *hypothesis automaton*. (This hypothesis often has a minimised state space.) For this it is necessary to have available a *structurally complete* (in an appropriate sense) set of membership queries and responses from A. For incremental learning, the goal is merely to construct a partial representation of A. This usually represents the "best" hypothesis about A using currently available membership data. The available data set may be structurally incomplete. When an incremental learning algorithm is guaranteed to eventually produce a behaviourally equivalent hypothesis automaton given sufficient data then it is said to *learn in the limit* (c.f. [28]).

For applications in software engineering such as test case generation and model mining, incremental learning is very efficient as we have shown in [45] and [43]. This is because: (i) real software systems are too large to be completely learned in practise, and (ii) incremental learning increases the opportunity to carry out on-the-fly model analysis by symbolic means such as model checking.

We can compare CGE with other well-known learning algorithms in the literature. In [23], an incremental version RPNI2 of the RPNI complete learning algorithm of [49] and [40] is presented. The RPNI2 algorithm has some features in common with our CGE algorithm. Most notably, both RPNI2 and CGE perform a recursive depth first search of a lexicographically ordered state set with backtracking. However, RPNI2 is explicitly coded for positive/negative outcomes only (language recognition), with no obvious generalisation to an arbitrary output set (Mealy). Furthermore, RPNI2, like most other automata learning algorithms, represents and manipulates the hypothesis automaton via a state partition set (inequalities). By contrast, CGE uses a more compact finite congruence generator set to represent the same information. Therefore, several expensive steps in RPNI2 such as quotient automaton derivation, deterministic partition merge, and compatibility (consistency) checking by parsing, become unnecessary in the CGE algorithm, or are replaced by an efficient alternative. Both RPNI2 and CGE learn in the limit. Neither algorithm needs to generate any new membership queries, other than those contained in the current query database, in order to produce a hypothesis automaton.

In [52] and [45] two different incremental modifications of the ID complete learning algorithm of [7] have been given. Again, these are explicitly coded for positive/negative outcomes (language recognition). Both use a *dead state* to model unknown information, and minimise membership queries between hypothesis formation. But since both algorithms use a tabular approach to partition construction, some membership queries are always required between each hypothesis automaton construction (albeit fewer than for L*.)

It is also worthwhile to compare CGE with complete learning algorithms. The well known L* algorithm of [8] is capable of constructing intermediate hypothesis automata before a structurally complete set of queries has been obtained. Thus L* has some incremental characteristics. Nevertheless, the interval between successive quotient hypothesis automata constructions can be very large, (of the order of tens or hundreds of thousands of queries) in practical situations.

4.3 String Rewriting Systems and Rule Completion

In this subsection we review the notion of a *string rewriting system* (SRS) as an efficient means of representing a finite congruence generator set. We will rely on the concept of a *confluent terminating* SRS, which has good computational properties. Such an SRS not only provides a compact representation of a set of congruence generators, it also provides a natural model of automaton simulation by *string rewriting*. This allows us to avoid an explicit graph-theoretic construction of the learned hypothesis automaton, if we so wish.

String rewriting is a special case of the more general theory of term rewriting ([21], [38]). We borrow many important ideas and results from this theory. One such concept is the *completion* of an SRS, which extends a non-confluent SRS to a confluent SRS. Our main result in this section is to define and prove the correctness of an efficient linear-time completion algorithm for SRS that we term *prefix completion*. Since completion is used so frequently during CGE learning, it is of great importance to have an efficient completion algorithm.

4.4.1. Definition.

(i) A *string rewriting rule* over Σ is a pair $(l,\ r) \in \Sigma^* \times \Sigma^*$. Often we use the more intuitive notation $l \to r$ to denote the rule $(l,\ r)$. Then l and r are termed the *left hand side* (lhs) and *right hand side* (rhs) of the rule, respectively. By a *string rewriting system* (SRS) over Σ we mean a set $R \subseteq \Sigma^* \times \Sigma^*$ of string rewriting rules.

(ii) Let $\rho = (l,\ r)$ be a string rewriting rule and let $\overline{\sigma},\ \overline{\sigma'} \in \Sigma^*$ be any strings. We say that $\overline{\sigma}$ *rewrites to* $\overline{\sigma'}$ using ρ and write $\overline{\sigma} \xrightarrow{\rho} \overline{\sigma'}$ if, and only if for some $\overline{\sigma_0} \in \Sigma^*$:

(ii.a) $\overline{\sigma} = l\ .\ \overline{\sigma_0}$ i.e. l is a prefix of $\overline{\sigma}$ and,

(ii.b) $\overline{\sigma'} = r\ .\ \overline{\sigma_0}$. So $\overline{\sigma'}$ is the result of syntactically replacing l by r in $\overline{\sigma}$.

(iii) If $R \subseteq \Sigma^* \times \Sigma^*$ is an SRS then we say that $\overline{\sigma}$ *rewrites to* $\overline{\sigma'}$ using R *in one step* and write $\overline{\sigma} \xrightarrow{R} \overline{\sigma'}$ if, and only if, for some $\rho \in R$, $\overline{\sigma} \xrightarrow{\rho} \overline{\sigma'}$. Thus $\xrightarrow{\rho} \subseteq \Sigma^* \times \Sigma^*$ and $\xrightarrow{R} \subseteq \Sigma^* \times \Sigma^*$ are binary relations.

(iv) We let $\xrightarrow{R^*} \subseteq \Sigma^* \times \Sigma^*$ denote the reflexive transitive closure of $\xrightarrow{R}$. Then $(\overline{\sigma},\ \overline{\sigma'}) \in \xrightarrow{R^*}$ if, and only if, there exists a finite sequence of strings $\overline{\sigma_0},\ \ldots,\ \overline{\sigma_k}$, for $k \geq 0$, and rules $\rho_i \in R$ for $0 \leq i < k$ such that

$$\overline{\sigma} = \overline{\sigma_0} \xrightarrow{\rho_0} \overline{\sigma_1} \xrightarrow{\rho_1} \ \ldots\ \xrightarrow{\rho_{k-1}} \overline{\sigma_k} = \overline{\sigma'}.$$

In this case we say that $\overline{\sigma}$ *rewrites to* $\overline{\sigma'}$ using R in finitely many steps and write $\overline{\sigma} \xrightarrow{R^*} \overline{\sigma'}$.

(v) Define the *bi-rewriting relation* $\xleftrightarrow{R^*} \subseteq \Sigma^* \times \Sigma^*$ for any $\overline{\sigma},\ \overline{\sigma'} \in \Sigma^*$, by: $\overline{\sigma} \xleftrightarrow{R^*} \overline{\sigma'} \ \Leftrightarrow$

$$\exists \overline{\sigma_0} \in \Sigma^* \text{ such that } \overline{\sigma} \xrightarrow{R^*} \overline{\sigma_0} \text{ and } \overline{\sigma'} \xrightarrow{R^*} \overline{\sigma_0}.$$

A string rewriting rule $l \to r$ can be viewed as an "oriented equation" between strings or as a single generator pair for a congruence on $I(\Sigma,\ \Omega)$. The rule $l \to r$ is asymmetric, and this asymmetry has a computational meaning in terms of syntactic replacement, as can be seen from Definition 4.1.(ii). Notice in this definition of string rewriting that we only allow matching and replacement of the lhs of a rule on *prefixes* of a string. Thus we might more correctly call this *prefix rewriting*. Prefix rewriting is therefore more restrictive than string rewriting using semi-Thue systems, (c.f. [21]), where one can match and rewrite anywhere within a string. However, prefix rewriting suffices for simulating automata (see Propositions 4.14 and 4.18 below) because of the existence of an initial state. For brevity, *rewriting* will always mean string rewriting in the sense of Definition 4.1.(ii).

Computation by string rewriting will mean repeated application of the rules of an SRS R to a string $\overline{\sigma}$. Two important issues arise here.

(i) Does the process of rewriting with R always *terminate*? That is, starting from any string $\overline{\sigma}$, do we eventually terminate, after executing finitely many rewrite steps using R with a string $\overline{\sigma'}$ to which no more rules from R can be applied? In this case we say that R is *terminating* and $\overline{\sigma'}$ is said to be *irreducible*.

(ii) Is string rewriting *deterministic*? That is, starting from any string $\overline{\sigma}$, is the irreducible result $\overline{\sigma'}$ of every terminating rewrite sequence using R *unique* and independent of the order of application of rules from R? Determinism has become known as *confluence* in the term rewriting literature (see below).

These two questions are in general difficult to answer. For semi-Thue systems the problems are known to be undecidable. However, in the special case of finite Mealy automata both questions can be answered in a positive way. Let us consider the termination problem first.

4.4.2. Definition.
(i) Let $D \subseteq \Sigma \times \Sigma$ be a linear ordering on Σ. Define the *lexicographic ordering* $\leq^D \subseteq \Sigma^n \times \Sigma^n$ for $n \geq 1$ inductively by

$$\sigma_1, \ldots, \sigma_n \leq^D \sigma'_1, \ldots, \sigma'_n \Leftrightarrow \sigma_1 \; D \; \sigma'_1 \;\; or$$

$$\sigma_1 = \sigma'_1 \;\; and \;\; \sigma_2, \ldots, \sigma_n \leq^D \sigma'_2, \ldots, \sigma'_n.$$

Then $\leq^D$ is a linear ordering on Σ^n induced by D.
(ii) We extend $\leq^D$ to the *short-lex ordering* $\leq^D \subseteq \Sigma^* \times \Sigma^*$ on strings by

$$\sigma_1, \ldots, \sigma_m \leq^D \sigma'_1, \ldots, \sigma'_n \Leftrightarrow m < n \;\; or$$

$$m = n \;\; and \;\; \sigma_1, \ldots, \sigma_m \leq^D \sigma'_1, \ldots, \sigma'_n.$$

Then $\leq^D$ is a well-ordering on all finite strings.

In the sequel we assume a fixed but arbitrarily chosen linear ordering D on Σ. Our purpose for introducing the short-lex ordering is to prove termination of rewrite sequences.

4.4.3. Definition. Let $l \to r \in \Sigma^* \times \Sigma^*$ be a rewrite rule. We say that $l \to r$ is *reducing* if, and only if, $r <^D l$. If $R \subseteq \Sigma^* \times \Sigma^*$ is an SRS then R is *reducing* if, and only if, every rule $\rho \in R$ is reducing.

4.4.4. Proposition. *Let $R \subseteq \Sigma^* \times \Sigma^*$ be a reducing SRS. Then R is terminating, i.e. there does not exist an infinite sequence $\overline{\sigma_0}, \overline{\sigma_1}, \ldots \in \Sigma^*$ of strings such that for each $i \geq 0$, there exists $\rho_i \in R$ for which $\overline{\sigma_i} \xrightarrow{\rho_i} \overline{\sigma_{i+1}}$.*

Let us now turn to question (ii) above, regarding the determinism of string rewriting. In general the order in which string rewriting rules are applied can indeed influence the outcome, i.e. the syntactic form of an irreducible string. An SRS which does not depend on the order of application of rules is termed a *confluent* SRS.

4.4.5. Definition. Let $R \subseteq \Sigma^* \times \Sigma^*$ be an SRS.

(i) We say that R is *confluent* if, and only if, for any $\overline{\sigma_0}, \overline{\sigma_1}, \overline{\sigma_2} \in \Sigma^*$, if $\overline{\sigma_0} \xrightarrow{R^*} \overline{\sigma_1}$ and $\overline{\sigma_0} \xrightarrow{R^*} \overline{\sigma_2}$ then there exists $\overline{\sigma} \in \Sigma^*$ such that $\overline{\sigma_1} \xrightarrow{R^*} \overline{\sigma}$ and $\overline{\sigma_2} \xrightarrow{R^*} \overline{\sigma}$.

(ii) We say that R is *locally confluent* if, and only if, for any $\overline{\sigma_0}, \overline{\sigma_1}, \overline{\sigma_2} \in \Sigma^*$, if $\overline{\sigma_0} \xrightarrow{R} \overline{\sigma_1}$ and $\overline{\sigma_0} \xrightarrow{R} \overline{\sigma_2}$ then there exists $\overline{\sigma} \in \Sigma^*$ such that $\overline{\sigma_1} \xrightarrow{R^*} \overline{\sigma}$ and $\overline{\sigma_2} \xrightarrow{R^*} \overline{\sigma}$.

Clearly confluence implies local confluence. The converse comes from a celebrated general result in term rewriting theory.

4.4.6. Newman's Lemma. *Let $R \subseteq \Sigma^* \times \Sigma^*$ be a reducing SRS. Then R is locally confluent if, and only if, R is confluent.*

Non-confluent (i.e. non-deterministic) SRS are undesirable. Fortunately, they can sometimes be converted to an equivalent confluent SRS by *completion*, i.e. conservatively adding extra rewrite rules that rectify rule divergence. This observation was pioneered by Knuth and Bendix in [39]. The Knuth-Bendix completion algorithm attempts to construct a confluent term rewriting system from an arbitrary (non-confluent) term rewriting system and an arbitrary well-ordering on terms.

Now, the Knuth-Bendix algorithm does not always terminate, since: (a) many word problems in algebra are undecidable, and (b) not every well-ordering has sufficient complexity to orient all critical pairs even for a decidable word problem. However, for specific rewriting systems and specific well-orderings we can find faster completion algorithms that can be guaranteed to terminate. We have derive one such linear-time completion algorithm for SRS, using the short-lex ordering, from first principles.

4.4.7. Prefix Completion Algorithm. Define the *prefix completion function* $C : \mathcal{P}^{fin}(\Sigma^* \times \Sigma^*) \to \mathcal{P}^{fin}(\Sigma^* \times \Sigma^*)$ constructively according to Algorithm 1.

Let $C(R)$ denote the rule set returned by the algorithm (if it terminates), given a finite input rule set $R \subseteq \Sigma^* \times \Sigma^*$. Also let $C_n(R)$ denote the set of rewrite rules obtained after the n-th iteration of the outermost while loop (if the loop reaches n iterations). It is understood that $C_0(R) = R$. We have been able to prove the following results about prefix completion in [42].

4.4.8. Proposition. *Let $R \subseteq \Sigma^* \times \Sigma^*$ be any SRS. If R is reducing then for each $n \geq 0$, $C_n(R)$ is a reducing SRS.*

4.4.9. Termination Theorem. *Let $R \subseteq \Sigma^* \times \Sigma^*$ be any finite reducing SRS. The prefix completion algorithm terminates given input R.*

4.4.10. Confluence Theorem. *Let $R \subseteq \Sigma^* \times \Sigma^*$ be any finite reducing SRS. Then $C(R)$ is a finite confluent reducing SRS.*

We have also analysed the run-time and space complexity of prefix completion. Consider any finite reducing SRS R, any maximal term $l \in Max_{\preceq}(R)$ and the tower of rules $T(l)$ under l. Define the height $height(T(l))$ of the tower $T(l)$ to be the sum of the lengths of all its blocks:

$$height(T(l)) = \sum_{1 \leq i \leq |T(l)|} |T(l)(i)|.$$

The Knuth-Bendix method of completion can add $(height(T(l)) - 1)^2/4$ additional rules when computing R^+, since every pair in $T(l)$ (adjacent or otherwise) can lead to a new rule. On the other hand, prefix completion adds at most $height(T(l)) + |T(l)|$ where $|T(l)|$ is total the number of blocks in $T(l)$, which is itself bounded by the height of $T(l)$. Thus Knuth-Bendix completion is an $O(n^2)$ algorithm for this problem while prefix completion is $O(n)$.

4.3.1 Learning by Consistent Generator Extension (CGE).

In this section we introduce the CGE learning algorithm for Mealy automata. The basic idea is to construct a sequence of hypothesis automata $H_0,\ H_1,\ \ldots$ using a series of queries $q_0,\ q_1,\ \ldots$ about an unknown Mealy automaton $\mathfrak{A}$. Each hypothesis automaton H_i is a quotient automaton:

$$H_i = I(\Sigma,\ \Omega)/\equiv^i, \quad i = 1,\ 2,\ \ldots,$$

and the sequence finitely converges to a quotient automaton H_n that is behaviorally equivalent to $\mathfrak{A}$, and has a minimised state space. Each congruence $\equiv^i$, for $i = 1,\ 2,\ \ldots$, is constructed from the current query set

$$\Lambda_i = \{\ q_0,\ q_1,\ \ldots,\ q_i\ \}.$$

and these sets are monotonically increasing, i.e. $\Lambda_i \subset \Lambda_{i+1}$. We can identify bounds on n and Λ_i which guarantee convergence, in terms of the size of $\mathfrak{A}$.

When comparing CGE with other automata inference algorithms in the literature, key features of CGE learning are the following:

1. **Function** $C(R)$
2. unfinished **:=** true
3. **while** unfinished **do**
4. Compute the set $Max_{\preceq}(R)$ of all maximal lhs terms;
5. **foreach** l $\in Max_{\preceq}(R)$ **do**
6. construct the tower $T(l)$ under l
7. unfinished **:=** false
8.
9. **foreach** l $\in Max_{\preceq}(R)$ **do**
10. //Process the first block in $T(l)$
11. //Collect all rhs terms from rules in block $T(l)(1)$
12. **for** j **:=** 2 **to** $|T(l)(1)|$ **do**
13. Rhs_Terms **:=** $Rhs\_Terms \cup \{\ T(l)(i)(j)_2\ \}$;
14. //Get the minimum rhs term in block $T(l)(1)$
15. rhs_min **:=** $T(l)(1)(1)_2$;
16. //Orient all critical pairs as new rules.
17. **foreach** $t \in Rhs\_Terms$ **do**
18. **if** $t \to rhs\_min \notin R$ **then** $R := R \cup \{\ t \to rhs\_min\ \}$
19. unfinished **:=** true
20.
21. //Process all remaining blocks in $T(l)$
22. **for** i **:=** 2 **to** $|T(l)|$ **do**
23. //Collect all rhs terms from rules in block $T(l)(i)$
24. **for** i **:=** 2 **to** $|T(l)(i)|$ **do**
25. Rhs_Terms **:=** $Rhs\_Terms \cup \{\ T(l)(i)(j)_2\ \}$;
26. //Complete the previous minimum rhs term
27. $rhs\_min\_completed$ **:=** $concat(rhs\_min, (T(l)(i)(1)_1 - T(l)(i-1)(1)_1))$;
28. //Compute new value of rhs_min and update Rhs_Terms
29. **if** $T(l)(i)(1)_2 \leq^D rhs\_min\_completed$ **then**
30. rhs_min **:=** $T(l)(i)(1)_2$
31. **if** $T(l)(i)(1)_2 \neq rhs\_min\_completed$ **then**
32. Rhs_Terms **:=** $Rhs\_Terms \cup \{\ rhs\_min\_completed\ \}$
33. **else**
34. rhs_min **:=** $rhs\_min\_completed$
35. Rhs_Terms **:=** $Rhs\_Terms \cup \{\ \ T(l)(i)(1)_2\ \}$
36. //Orient all critical pairs as new rules
37. **foreach** $t \in Rhs\_Terms$ **do**
38. **if** $t \to rhs\_min \notin R$ **then** $R := R \cup \{\ t \to rhs\_min\ \}$
39. $unfinished$ **:=** $true$
40.
41. **return** R
42. **end**

Algorithm 1: Prefix Completion Algorithm

(i) No explicit state space partition is used (e.g. a table), nor is there any use of an equivalence oracle.

(ii) A new hypothesis automaton H_{i+1} can be constructed after each individual new query q_{i+1}.

(iii) There is complete freedom to choose each new query q_{i+1}.

(iv) The hypothesis automaton H_i is never explicitly constructed. Instead a finite generator set for the congruence $\equiv^i$ is returned.

Features (ii) and (iii) are an important advantage of computing with congruence generators rather than an explicit state space partition. In addition, using a congruence generator set gives a more compact hypothesis representation than explicit automaton construction.

The main computational problem is to construct a confluent terminating SRS R^i representing the state congruence $\equiv^i_Q$. We can construct R^i from Λ_i incrementally, starting from the empty SRS $R^i_0 = \emptyset$ by considering every possible reducing rewrite rule ρ_j for $j = 0, \ldots, l$ (up to a certain finite limit l). For each such rule ρ_j, we analyse whether ρ_j can be consistently added to the current SRS R^i_j to obtain a larger SRS R^i_{j+1}. Therefore we term this process *consistent generator extension*.

Let us begin by formalising the appropriate notion of consistency. Consistency applies to both congruences and automata. A consistent congruence (or SRS), is one that does not contradict the information within a set Λ of queries on $\mathfrak{A}$. On the other hand, an automaton $\mathfrak{A}$ is consistent if it does not identify outputs which are syntactically distinct.

4.5.1. Definition.

(i) Let $\equiv \subseteq (\Sigma^+ \cup \Omega)^2$ be an equivalence relation on outputs. We say that $\equiv$ is *consistent* if and only if, for any output symbols ω, $\omega' \in \Omega$

$$\omega \neq \omega' \Rightarrow \omega \not\equiv \omega'.$$

(ii) Let $\equiv \subseteq \Sigma^* \times \Sigma^*$ be any relation and $\Lambda \subseteq \Sigma^+ \times \Omega$ be any set of queries. Then $\equiv$ is said to be Λ-*consistent* if, and only if $\equiv^\Lambda$ is consistent (c.f. Definition 4.16).

(iii) Let $R \subseteq \Sigma^* \times \Sigma^*$ be a confluent reducing SRS then R is said to be Λ-*consistent* if and only if, the bi-rewriting relation $\overset{R^*}{\longleftrightarrow}$ is Λ-consistent.

(iv) Let $\mathfrak{A} \in MA(\Sigma, \Omega)$ be any automaton. Then $\mathfrak{A}$ is said to be *consistent* if, and only if, for any output symbols ω, $\omega' \in \Omega$

$$\omega \neq \omega' \Rightarrow \omega_{\mathfrak{A}} \neq \omega'_{\mathfrak{A}}.$$

Since inconsistent SRS should always be avoided during learning, we give an explicit algorithm for checking consistency of a new rewrite rule with an existing SRS and a set of queries.

4.5.2. Consistency Algorithm. Define the *consistency function* $Cons : \mathcal{P}^{fin}(\Sigma^* \times \Sigma^*) \times \mathcal{P}^{fin}(\Sigma^+ \times \Omega) \times \Sigma^* \times \Sigma^* \to \{ true, false \}$ constructively by:

$Cons\ (R, \Lambda, , \overline{\sigma}, \overline{\sigma'}) =$

let $S = C(\ R \cup \{\ \overline{\sigma} \to \overline{\sigma'}\ \}\ )$ in

$$\begin{cases} false & \text{if } Norm_S(\sigma_1, \ldots, \sigma_m) = Norm_S(\sigma'_1, \ldots, \sigma'_n), \\ & \text{for some } (\sigma_1, \ldots, \sigma_m, \sigma, \omega\ ) \in \Lambda \\ & \text{and } (\sigma'_1, \ldots, \sigma'_n, \sigma, \omega'\ ) \in \Lambda \text{ with } \omega \neq \omega'. \\ \\ true & \text{otherwise,} \end{cases}$$

for any SRS $R \subseteq \Sigma^* \times \Sigma^*$, query set $\Lambda \subseteq \Sigma^+ \times \Omega$ and strings $\overline{\sigma}$, $\overline{\sigma'} \in \Sigma^*$.

4.5.3. Correctness Theorem. *Let $R \subseteq \Sigma^* \times \Sigma^*$ be a finite confluent reducing SRS and let $\Lambda \subseteq \Sigma^+ \times \Omega$ be any set of queries. For any $\overline{\sigma},\ \overline{\sigma'} \in \Sigma^*$ such that $\overline{\sigma'} <^D \overline{\sigma}$,*

$$Cons\ (R,\ \Lambda,\ ,\ \overline{\sigma},\ \overline{\sigma'}) = false$$

if, and only if $S = C(\ R \cup \{\ \overline{\sigma} \to \overline{\sigma'}\ \}\ )$ is a Λ-inconsistent SRS.

With the prefix completion algorithm and the above consistency algorithm as auxiliary algorithms, we can now present the CGE algorithm for incremental learning of Mealy automata.

The CGE algorithm takes as input a set Λ of queries, an SRS R which is under construction, a finite sequence $A = A_1,\ \ldots,\ A_l$ of strings $A_i \in \Sigma^*$ (state names) which is strictly linearly ordered by $<^D$, and two indices $m,\ n$ within $1,\ \ldots,\ l$ for the lhs and rhs of a putative reducing rewrite rule $A_m \to A_n$. The indices m and n are recursively incremented to traverse A in a specific order, starting from their initial values. The basic idea is first to check that the rule $A_m \to A_n$ is not already subsumed by other rules in R (in which case it would be redundant to add it). Assuming that it is not already subsumed, then we add this rule $A_m \to A_n$ to R and complete. If the resulting confluent reducing SRS $C(\ R \cup \{\ A_m \to A_n\ \}\ )$ is Λ-consistent then the rule is retained, otherwise it is discarded. In both cases, we recursively proceed to the next rule in A until we have traversed all of A. The strict linear ordering of A by $<^D$ ensures that $A_m \to A_n$ is always reducing whenever $m > n$.

4.5.4. CGE Learning Algorithm. Let $\Lambda \subseteq \Sigma^+ \times \Omega$ be a given finite set of queries. Define the *Consistent Generator Extension function*

$$CGE_\Lambda : \mathcal{P}^{fin}(\Sigma^* \times \Sigma^*) \times (\Sigma^*)^+ \times \mathbb{N} \times \mathbb{N} \to \mathcal{P}^{fin}(\Sigma^* \times \Sigma^*)$$

constructively for any SRS $R \subseteq \Sigma^* \times \Sigma^*$, any finite sequence $A = A_1,\ \ldots,\ A_l \in (\Sigma^*)^+$ of strings, and any indices $m,\ n \in \mathbb{N}$, by Algorithm 2.

4.5.5. Definition. Let $\Lambda \subseteq \Sigma^+ \times \Omega$ be any finite set of queries and let $A \in (\Sigma^*)^+$ be any finite sequence of strings. Define the *consistent generator extension function*

$$CGE_\Lambda(A) = CGE_\Lambda(\ \emptyset,\ A,\ 2,\ 1\ ).$$

Now we must consider the eventual convergence of the sequence of hypothesis automaton generated by the CGE algorithm (learning in the limit). We can show that after sufficiently many queries are made on $\mathfrak{A}$ the CGE algorithm produces a hypothesis automaton that is behaviourally equivalent with $\mathfrak{A}$. The key insight for proving convergence is to establish that all loops in the graph structure of the unknown automaton $\mathfrak{A}$ will be correctly learned by executing a sufficient number of queries on $\mathfrak{A}$. This number is bounded by the size of the largest loop in $\mathfrak{A}$. Using this fact, we can establish that the CGE algorithm eventually correctly learns the unknown automaton $\mathfrak{A}$ up to behavioural equivalence. Let $\equiv^{\mathfrak{A}}$ be the kernel of the unique homomorphism $\phi^{\mathfrak{A}} : I(\Sigma,\ \Omega) \to \mathfrak{A}$. Comparing the learned output congruence $\overset{R^*\Lambda}{\longleftrightarrow}$ with $\equiv^{\mathfrak{A}}_{\Omega}$ we need to show that CGE learning is:
(i) *sound* i.e. $\overset{R^*\Lambda}{\longleftrightarrow} \subseteq \equiv^{\mathfrak{A}}_{\Omega}$, and no incorrect behaviours are learned, and
(ii) *complete* i.e. $\equiv^{\mathfrak{A}}_{\Omega} \subseteq \overset{R^*\Lambda}{\longleftrightarrow}$, and all correct behaviours are learned.

4.5.6. Soundness Theorem. *Let $\mathfrak{A} \in MA(\Sigma,\ \Omega)$ be a consistent automaton. Let n be the length of the longest acyclic path in $\mathfrak{A}$. Let $\Lambda \subseteq \Sigma^+ \times \Omega$ be any set of queries on $\mathfrak{A}$ that contains all queries of length $2n+1$ or less. Let $R = CGE_\Lambda(\Lambda^+_{<^D}\ )$.*

(1) For all $(l,\ r) \in \Sigma^* \times \Sigma^*$

$$l \overset{R^*}{\longleftrightarrow} r \Rightarrow \text{for all } \overline{\sigma} \in \Sigma^+\ l\,\overline{\sigma}\ \equiv^{\mathfrak{A}}_{\Omega}\ r\,\overline{\sigma}.$$

(2) $\overset{R^*\Lambda}{\longleftrightarrow} \subseteq \equiv^{\mathfrak{A}}_{\Omega}$.

1. **Function** $CGE_\Lambda(R, A, m, n)$
2. //Check subsumption and consistency
3. //of rule $A_m \rightarrow A_n$
4. **if** $A_m \overset{R^*}{\not\longrightarrow} A_n$
5. **and** *Cons* ($R,\ \Lambda,\ A_m,\ A_n$) **then**
6. //Add rule $A_m \rightarrow A_n$ to R
7. **if** $n = m - 1$ **and** $m < |A|$ **then**
8. **return** $CGE_\Lambda($ $C($ $R \cup \{\ A_m \rightarrow A_n\ \}$),
9. $A,\ m + 1,\ 1$)
10. **else if** $n < m - 1$ **then**
11. **return** $CGE_\Lambda(C(R \cup \{\ A_m \rightarrow A_n\ \}),$
12. $A,\ m,\ n + 1)$
13. **else**
14. //Finished traversal of A
15. **return** $C($ $R \cup \{\ A_m \rightarrow A_n\ \}$)
16. **else**
17. //Don't add rule $A_m \rightarrow A_n$ to R
18. **if** $n = m - 1$ and $m < |A|$ **then**
19. **return** $CGE_\Lambda($ $R,\ A,\ m + 1,\ 1$)
20. **else if** $n < m - 1$ **then**
21. **return** $CGE_\Lambda($ $R,\ A,\ m,\ n + 1$)
22. **else**
23. //Finished traversal of A
24. **return** R
25. **end**

Algorithm 2: CGE Algorithm

The converse of Soundness Theorem 4.5.6 is the following.

4.5.7. Completeness Theorem. *Let $\mathfrak{A} \in MA(\Sigma, \Omega)$ be a consistent automaton. Let n be the length of the longest acyclic path in $\mathfrak{A}$. Let $\Lambda \subseteq \Sigma^+ \times \Omega$ be any set of queries on $\mathfrak{A}$ that contains all queries of length $2n+1$ or less. Let $R = CGE_A(\Lambda^+_{<^D})$, then*

$$\equiv^{\mathfrak{A}}_{\Omega} \subseteq \overset{R^*\Lambda}{\longleftrightarrow}.$$

Input: A sequence $S = (\overline{\sigma}_1, \omega_1), \ldots, (\overline{\sigma}_n, \omega_n)$ of n queries of the I/O behavior of $\mathfrak{A}$, where $(\overline{\sigma}_i, \omega_i) = (\sigma_i^1, \ldots, \sigma_i^{k(i)}, \omega_i) \in \Sigma^+ \times \Omega$.
Output: A sequence $(R_i^{state}, R_i^{output})$ for $i = 1, \ldots, n$ of congruence generator sets for hypothesis machines M_i represented as SRS.

1. **begin**
2. //Perform initialization
3. $\Lambda = \emptyset$, $\Lambda^R = \emptyset$, $R = \emptyset$, $A = \emptyset$, $i = 1$,
4. **while** $i \leq n$ **do**
5. //Normalise the i-th query and all its prefix queries with R
6. $norm = Norm_R(\ PrefQuery((\overline{\sigma}_i, \omega_i))\ )$
7.
8. **if** $norm \not\subseteq \Lambda^R$ **then**
9. //At least one prefix query of i-th query
10. //$(\overline{\sigma}_i, \omega_i)$ has no equivalent in Λ^R.
11. //So update Λ, R and Λ^R.
12. //This will also resolve inconsistency
13. //if a prefix of $(\overline{\sigma}_i, \omega_i)$ is inconsistent with R.
14. $\Lambda = \Lambda \cup \{\ PrefQuery((\overline{\sigma}_i, \omega_i))\ \}$
15. $A = lhs(\Lambda) \cup$
16. $\{\overline{\tau}\sigma_0 \mid \exists\, \overline{\sigma} \in lhs(\Lambda),\ \overline{\tau} \prec \overline{\sigma},\ \sigma_0 \in \Sigma\ \}$
17. $R = CGE_\Lambda(\ A_{<^D}\ )$
18. $\Lambda^R = Norm_R(\Lambda)$
19.
20. Print(i; $(\overline{\sigma}_i, \omega_i)$;
21. $(R, \{\ l\sigma \to r\sigma \mid l \to r \in R,\ \sigma \in \Sigma\ \} \cup \Lambda^R)\ )$
22. $i = i + 1$

Algorithm 3: CGE Iteration Algorithm

It only remains to insert the CGE function into an iterative loop where it computes successive hypothesis automata, and this is defined in Algorithm 3. In this algorithm, new queries are repeatedly read in, and for each new query a new hypothesis automaton is constructed.

4.3.2 A Case Study

To give some feeling for the capabilities of CGE learning, we will describe a typical case study.

The Transmission Control Protocol (TCP) is a widely used transport protocol over the Internet. We present here a performance evaluation of our LBT architecture applied to testing a simplified model of the

TCP/IP protocol as the 11 state EMA shown in Figure 4.1. This example involves an input alphabet of 12 symbols and an output alphabet of 6 symbols. Note that self loops on null transitions are omitted in this diagram. A complete diagram must have $11 * 12 = 132$ transitions. This Mealy machine can be learned in approximately 7 minutes, using approximately 88,000 queries. This performance is somwhat similar (i.e. within the same order of magnitude) to the performance of existing DFA learning algorithms.

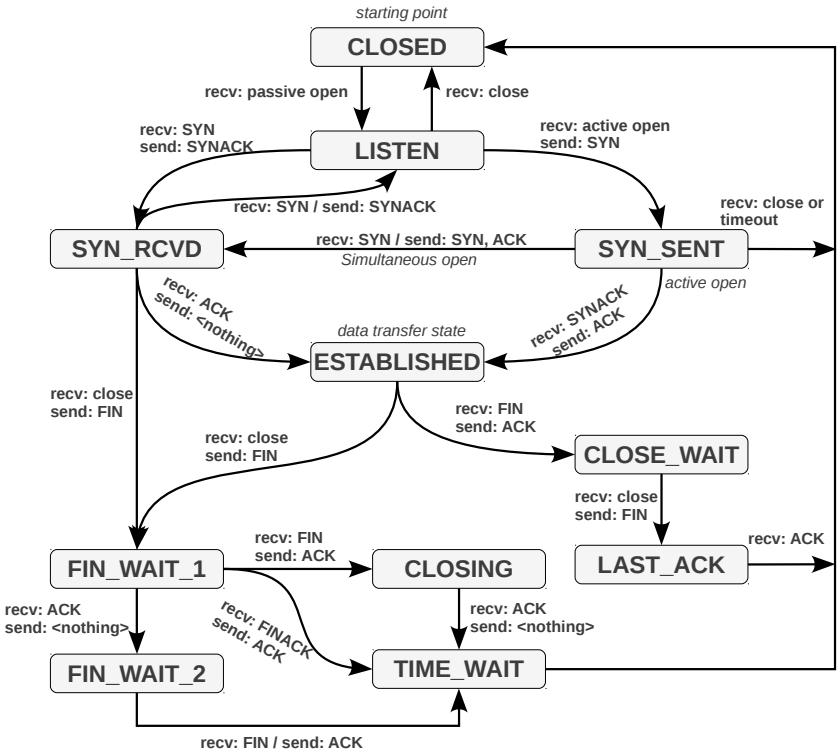

Figure 4.1: TCP Mealy Machine Model

Chapter 5

Conclusion

In this deliverable we have studied three distinct, but related approaches to model mining. Model mining is a new idea in relation to software product lines, but at the same time it builds on well-established ideas in computer science such as static analysis, semantics, and automata theory.

The relationships between the approaches we have studies can be depicted as in Figure 5.1. In the figure

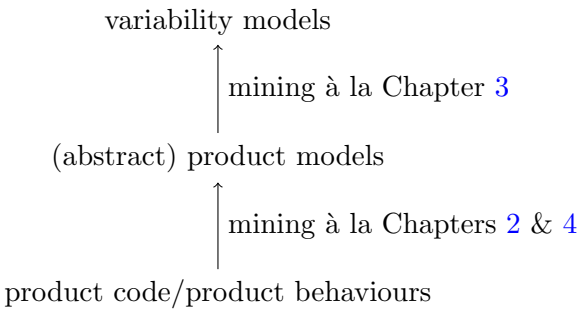

Figure 5.1: Relationship between the model mining approaches.

an arrow indicates a model mining step, using one of the approaches, mining the target from the source. Two of the approaches in this deliverable go from product code or product behaviour to product models. Which approach to use must be determined by how to treat a system: as structured code (Chapter 2) or as a black box with observable behaviour (Chapter 4). If there are several related product models, then these may be mined to yield a variability model (Chapter 3), provided the concrete product descriptions can be abstracted into a specific form.

In our research we have used mined models for resource or termination analysis, for making explicit shared and variable parts, and for testing programs. In addition, the obtained models have a value as-such for developers since mined models describe the inputs in an abstract and compact manner, facilitating overview.

5.1 Future Work

The field of model mining for software product lines is new and presents lots of possibilities. Future work in this area should combine both unifying and relating the three approaches presented in this deliverable, and studying new approaches, either complementary or generalising the three. We go on to remark further work in the context on each approach.

We would like to extend JMS2ABS to dealing with more complex distribution model of JMS, including adapting the underlying ABS model of JMS. The ABS' mechanisms are not very different from those used by other distributed object-based modelling languages [32, 33], and so we expect our work to be useful conclusions beyond the mere case study performed in Chapter 2.

Future work on variability models will focus on the practical evaluation of the proposed method for variability model mining, considering in particular sets of (legacy code) products that have not been designed as a family from the outset. Further effort is planned on generalising the model with optional and multiple variant selections and with requires/excludes constraints between variants, and on adapting accordingly the model reconstruction transformation.

The overwhelming problem in automaton learning theory is the problem of dealing with large models. This often prevents industrial uptake of the methods. Therefore we will primarily focus on improving scalability of the current learning algorithms. The Kripke structure learning algorithm for finite data types can be further improved by applying a state space minimisation algorithm to the product automaton that is currently output. This would greatly speed up subsequent operations such as model checking. We will look at generalising Hopcroft's $n \log n$ minimisation algorithm from DFA to Kripke structures, which would give an efficient and scalable solution to minimisation. The current CGE learning algorithm has rather poor performance considering its theoretical potential. One reason is that it simultaneously combines learning with minimisation in a rather inefficient way. We will look at learning non-minimal representations (which are space efficient using congruence representations). These can be based on inferring state loops only, and ignoring all other state merges. Minimisation could then be more efficiently applied as a once-only operation, after learning has converged. Finally, we could consider producing output in explicit ABS format, to support compatibility with other HATS tools.

Bibliography

[1] Report on Resource Guarantees, March 2011. Deliv. 4.2 of project FP7-231620 (HATS), available at http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable42.pdf.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD Conference*, pages 207–216, 1993.

[3] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, and G. Puebla. Cost Analysis of Concurrent OO programs. In *APLAS'11*, volume 7078 of *Lecture Notes in Computer Science*. Springer, 2011.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.*, 413, January 2012.

[5] E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating Concurrent Behaviors with Worst-Case Cost Bounds. In *FM 2011*, volume 6664 of *Lecture Notes in Computer Science*. Springer, 2011.

[6] Vander Alves, Christa Schwanninger, Luciano Barbosa, Awais Rashid, Peter Sawyer, Paul Rayson, Christoph Pohl, and Andreas Rummler. An exploratory study of information retrieval techniques in domain analysis. In *SPLC*, pages 67–76, 2008.

[7] D. Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, October 1981.

[8] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(1):87–106, November 1987.

[9] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model Superimposition in Software Product Lines. In *International Conference on Model Transformation (ICMT '09)*, volume 5563 of *LNCS*, pages 4–19. Springer, 2009.

[10] A. W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33:17–20, April 1998.

[11] J.L. Balcazar, J. Diaz, and R. Gavalda. Algorithms for learning finite automata from queries. In *Advances in Algorithms, Languages and Complexity*, pages 53–72. Kluwer, 1997.

[12] R. Baldoni, R. Beraldi, S. Tucci Piergiovanni, and A. Virgillito. On the modelling of publish/subscribe communication systems. *Concurr. Comput. : Pract. Exper.*, 17:1471–1495, October 2005.

[13] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[14] A. Bertolino, A. Calabro, M. Merten, and B. Steffen. Never-stop learning: Continuous validation of learned models for evolving systems through monitoring. *ERCIM News*, 88:28–29, 2012.

[15] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *GPCE*. Springer, 2010.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude—A High-Performance Logical Framework*. Number 4350 in Lecture Notes in Computer Science. Springer, 2007.

[17] Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Generative Programming and Component Engineering (GPCE '05)*, volume 3676 of *LNCS*, pages 422 – 437. Springer, 2005.

[18] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[19] Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. Sample spaces and feature models: There and back again. In *SPLC*, pages 22–31, 2008.

[20] C. de la Higuera. *Grammatical Inference*. Cambridge University Press, 2010.

[21] N. Dershowitz and J-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*. North Holland, 1990.

[22] Ina Schaefer Dilian Gurov, Bjarte M. Østvold. A hierarchical variability model for software product lines. Technical Report TRITA-CSC-TCS 2011:1, KTH, September 2011.

[23] P. Dupont. Incremental regular inference. In *Proceedings of the Third ICGI-96*, number 1147 in LNAI, 1996.

[24] M. Broy et al. (eds). *Model-based Testing of Reactive Systems, LNCS 3471*. Springer, 2005.

[25] P. Eugster, P. Felber, R. Guerraoui, and AM. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, June 2003.

[26] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1996.

[27] D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Proceedings of SPIN'03*. Springer-Verlag, 2003.

[28] E.M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[29] H. Gomaa. *Designing Software Product Lines with UML*. Addison Wesley, 2004.

[30] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51:1409–1427, October 2009.

[31] A. Haber, H. Rendel, B. Rumpe, I. Schaefer, and F. van der Linden. Hierarchical variability modeling for software architectures. In *Software Product Line Conference (SPLC 2011)*, 2011. (to appear).

[32] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[33] P. Haller and F. Sommers. *Actors in Scala: Concurrent programming for the multi-core era*. Artima, PrePrint edition, March 2011.

[34] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. *Java Message Service Specification*. Sun Microsystems, Inc., Version 1.1. April 2002.

[35] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Software Product Line Conference (SPLC '08)*, pages 139–148. IEEE, 2008.

[36] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proceedings of FMCO'10*, volume 6957 of *Lecture Notes in Computer Science*. Springer-Verlag, 2011. To appear.

[37] S. Katsumata and A. Ohori. Proof-directed de-compilation of low-level code. In *ESOP*, volume 2028 of *Lecture Notes in Computer Science*. Springer, 2001.

[38] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science: Volume 2*, pages 2–117. Oxford University Press, 1992.

[39] D.E. Knuth and P. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–269. Pergamon Press, 1970.

[40] K.J. Lang. Random dfa's can be approximately learned from sparse uniform examples. In *Fifth ACM Workshop on Computational Learning Theory*, pages 45–52. ACM Press, 1992.

[41] Felix Loesch and Erhard Ploedereder. Optimization of variability in software product lines. In *SPLC*, pages 151–162, 2007.

[42] K. Meinke. Cge: A sequential learning algorithm for mealy automata. In *Proc. Tenth Int. Colloq. on Grammatical Inference (ICGI 2010)*, number 6339 in LNAI, pages 148–162. Springer, 2010.

[43] K. Meinke and F. Niu. Learning-based testing for reactive systems using term rewriting technology. In *Proc. 23rd IFIP Int. Conf. on Testing Software and Systems (ICTSS 2011)*, number 7019 in LNCS, pages 97–114. Springer, 2011.

[44] K. Meinke and M. Sindhu. Correctness and performance of an incremental learning algorithm for finite automata. Technical report, School of Computer Science and Communication, Royal Institute of Technology, Stockholm, 2010.

[45] K. Meinke and M. Sindhu. Incremental learning-based testing for reactive systems. In *Proc Fifth Int. Conf. on Tests and Proofs (TAP2011)*, number 6706 in LNCS, pages 134–151. Springer, 2011.

[46] K. Meinke and J.V. Tucker. Universal algebra. In *Handbook of Logic in Computer Science: Volume 1*, pages 189–411. Oxford University Press, 1993.

[47] Nan Niu and Steve Easterbrook. Concept analysis for product line requirements. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 137–148, 2009.

[48] N. Noda and T. Kishi. Aspect-Oriented Modeling for Variability Management. In *Software Product Line Conference (SPLC '08)*, pages 213–222. IEEE, 2008.

[49] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, Series in Machine Perception and Artificial Intelligence. World Scientific, 1992.

[50] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Termination Analysis of Java Bytecode by Term Rewriting. In Johannes Waldmann, editor, *International Workshop on Termination WST'09*, Leipzig, Germany, June 2009.

[51] R. Parekh and V. Honavar. Grammar inference, automata induction and language acquisition. In *Handbook of Natural Language Processing*. Marcel Dekker, 2000.

[52] R.G. Parekh, C. Nichitiu, and V.G. Honavar. A polynomial time incremental algorithm for regular grammar inference. In *Proc. Fourth Int. Colloq. on Grammatical Inference (ICGI 98)*, LNAI. Springer, 1998.

[53] Jennifer Pérez, Jessica Díaz, Cristóbal Costa Soria, and Juan Garbajosa. Plastic Partial Components: A solution to support variability in architectural components. In *WICSA/ECSA*, pages 221–230, 2009.

[54] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques.* Springer, 2005.

[55] M. Richards, R. Monson-Haefel, and D. A. Chappell. *Java Message Service - Creating Distributed Enterprise Applications (2. ed.).* O'Reilly, 2009.

[56] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. Automatic variation-point identification in function-block-based models. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 23–32, New York, NY, USA, 2010. ACM.

[57] I. Schaefer, D. Gurov, and S. Soleimanifard. Compositional algorithmic verification of software product lines. In *Postproceedings of Intl. Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 184–203. Springer, 2011.

[58] Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Trans. Softw. Eng. Methodol.*, 5:146–189, April 1996.

[59] M. Utting and B. Legeard. *Practical Model-Based Testing.* Morgan Kaufmann, 2007.

[60] R. van Ommering. Software reuse in product populations. *IEEE Trans. Software Eng.*, 31(7):537–550, 2005.

[61] M. Völter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Software Product Line Conference (SPLC '07)*, pages 233–242. IEEE, 2007.

[62] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9), 1975.

[63] T. Ziadi, L. Hélouët, and J. Jézéquel. Towards a UML Profile for Software Product Lines. In *Software Product Familiy Engineering (PFE '03)*, volume 3014 of *LNCS*, pages 129–139. Springer, 2003.

Glossary

Abstract data type (ADT) A data type defined by operations on the type and where the implementation of the type is hidden.

Hierarchical variability model A variability model composed by putting together (sub) variability models and possibly other constructs, meaning that models written in the same language occur at different "levels" according to the composition.

Java Message Service (JMS) An industry standard for message communication in Java enterprise systems, including publish/subscribe communication, see also *publish/subscribe system*.

Intermediate representation (IR) A language more suitable for analysis and reasoning that machine code or byte code. For example, IR could have structured control flow instead of labels and jump instructions.

Mealy machine A form of automaton model where outputs are associated with transitions rather than states. See also *Moore machine*.

Moore machine A form of automaton model where outputs are associated with states rather than transitions. See also *Mealy machine*.

Problem space variability Variation described in term of features where a feature is a user-visible product characteristic. See also *solution space variability*.

Publish/subscribe system A (large-scale) distributed system with a loosely coupled form of interaction between system parts, these parts being publishers sending messages to (registered) subscribers via topics.

Solution space variability Variation described in terms of artifacts where the artifacts are used to implement the actual products. See also *problem space variability*.

Simple hierarchical variability model A hierarchical variability model satisfying certain desirable properties, including uniqueness with respect to a particular set of products.

Variability model An abstract description of how the products of a software product line may vary.