Formal Modelling and Verification of Trust in a Pervasive Application

(Deliverable WP4-01)

June 2004

Trusted Software Agents and Services for Pervasive Information Environments project

DTI Next Wave Technologies and Markets Programme

QinetiQ

University of Southampton

Editor

Stéphane Lo Presti University of Southampton

Other contributors

University of Southampton:
Michael Butler
Michael Leuschel
Colin snook
Phillip Turner

Contact

Stéphane Lo Presti
School of Electronics and Computer Science
University of Southampton
SOUTHAMPTON SO17 1BJ
United Kingdom

Email: splp@ecs.soton.ac.uk
Web: http://www.ecs.soton.ac.uk/~splp/
Phone: +44 (0)23 8059 7684
Fax: +44 (0)23 8059 3045

Record of changes

Issue	Date	Detail of changes
1.0	25/06/04	First Draft submitted to DTI.

Executive summary

This report is deliverable WP4-01 of the project “Trusted Software Agents and Services for Pervasive Information Environments.” The deliverable reports on the activities of formal modelling and verification of a pervasive application which follows from previous results in the project.

The pervasive application is based on several pervasive scenarios already devised and is centred on the user location. This location-based system is first architecturally simplified, while trust requirements are derived from the Trust Analysis Framework presented in the deliverable WP2-01.

This first abstraction is then completed by formal modelling of the system in the B formal method. These models enable us to clarify the decision decisions leading to fulfil the trust requirements. We show that the system policy structure is influenced by the priorities given to the system operations and that a sufficiently high level of abstraction is required to model trust properties.

The modelling activity is completed by formal verification using the ProB model-checker to automate part of this process. Several models are checked successfully, while detection of errors in other models enables us to understand better the behaviour of the system. In particular, issues relative to the dynamicity of modelled elements are highlighted.

The overall methodology followed during these activities proved useful at helping us specifying accurately the trust requirements, so that the pervasive application can be completed in consequence, and is as follows:

1) Model important features of the system
   First vaguely type the variables; then write a set of operations corresponding to complementary features while (possibly) modifying the variable types to ease this writing; consider the variables by group of similar dynamic properties;
2a) Model-check the model
   2a.a) Property violation detected
      Examine the various aspects of the model (variables, enabled operations, history of operations) to see what part of the property is “false”; Correct the model accordingly;
   2a.b) No property violation detected
      Go back to 2a until coverage rate is enough; possible changes to the model include: modify the initialisation to test other situations (in B use “choice by predicate”); add inconsistencies in the model;
2b) Animate the model
   2b.a) Execute the desired sequence of operations (validation);
   2b.b) Find an interesting state, then 2a.a or 2a.b applies;
   2b.c) Backtrack from a state where the invariant is violated;
3) Go back to 1 (complete the model) or refine the model.

Table of Contents

1 Introduction

This deliverable WP4-01 of the project “Trusted Software Agents and Services for Pervasive Information Environments” reports on the formal modelling of a pervasive application that implements ideas developed in previous deliverables of the project, and the verification of the models created using formal tools.

1.1 Background

Pervasive Computing is a new paradigm that places the user at the centre of a system made of computing devices embedded in physical objects. It is materialised by the convergence of small footprint computing platforms (PDA, mobile phones), next-generation wireless networks (UMTS, Wi-Fi, Bluetooth) and new modalities of human-machine interaction where trust and security play a crucial role.

Trust has recently been the focus of much effort, as much for research scientists [3] as for application developers, with growing concerns on security and privacy. Our initial investigation into the trust issues in pervasive environment [5] concluded that a Trust Analysis Grid could be used to analyse the trust issues in a variety of pervasive scenarios. For further details, the reader should refer to the deliverable WP2-01 [5].

1.2 Aim of the project

This project aims to identify the critical trust issues in pervasive computing, and to exploit formal techniques for validating the trustworthiness of emerging agent, Semantic Web and Grid technologies to support pervasive systems.

The approach taken to address this aim is to:

1. Investigate issues of trust, such as security, dependability and privacy, within the context of software agents and services in pervasive information environments;
2. Develop strategies to introduce trust in these systems, underpinned by formal validation techniques;
3. Build small-scale realistic demonstrators that highlight the problems and solutions in the healthcare domain, with a view towards eventual productisation.

1.3 Aim of the report

This report addresses step 2 of the approach chosen for this project and aims at presenting the formal modelling and verification of models of a pervasive application implemented by QinetiQ. We will present the various methods and tools that were used to model the pervasive system, focusing on the important features from a design and implementation point of view. The formal verification of these models provides us with interesting results on the way to introduce trust in the system. The methodology used throughout the modelling and verification activities are summarised in a set of guidelines.

1.3.1 Structure of the document

Section 2 describes the pervasive application under consideration. Section 3 describes the various models corresponding to the different possible cases. Section 4 finally presents the results of verification of the models using formal tools.

2 Modelled Pervasive Application

This Section describes the pervasive application that is under consideration in this report. It is the starting point of the efforts and the results that this report delivers. We start from a description of the origin and architecture of the system, and then select the important element that will drive the remaining of our work.

2.1 Introduction

The first deliverable of the project specified five pervasive scenarios that were validated by domain experts. All of them shared common trust issues that were highlighted and analysed in the deliverable WP2-01 [5].

The various pervasive systems described in these scenarios have in common the use of services based on the location of the user, a feature that is representative of pervasive systems currently designed and implemented and, thus, that represents an interesting and important application to model and verify.

The location-based system has been implemented by QinetiQ in Java and for Pocket PC PDAs (the pervasive devices). It uses the Wi-Fi (IEEE 802.11) communication technology and exports some services (e.g. exchange of images between users) as Web Services (WS). The Wi-Fi technology is used in indoor environments and corresponds to the situation of most of our pervasive scenarios. The Royal Glamorgan Hospital (Llantrisant, Wales) has been contacted to trial the prototype in their working environment.

2.2 Architecture

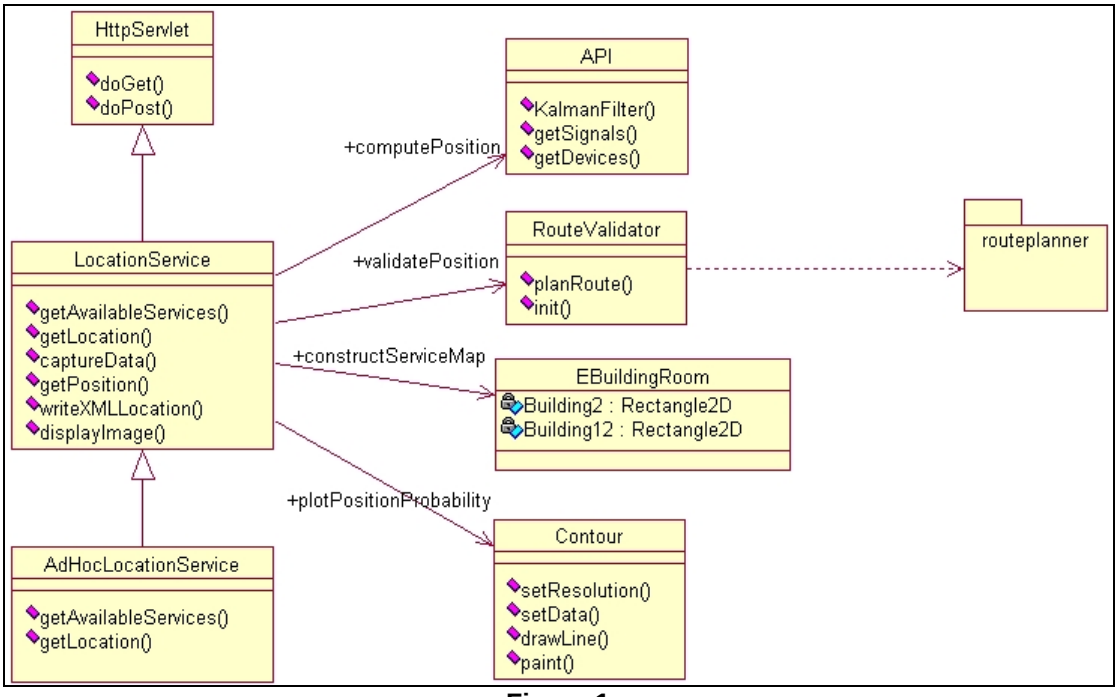

Figure 1

The location-based system is organised around a `LocationService` object that runs on a Web server (i.e. class `HttpServlet`), as the UML diagram shown in Figure 1 illustrates. The object is initialised by the capture of data coming from wireless access points, so that a map of the various rooms can be created from the Wi-Fi signal strengths, and by the declaration of the various services offered. Then, the `LocationService` object waits for

PDA requesting, either its location, or the set of currently available services around the PDA. The location is first estimated (position with associated probability) from the current Wi-Fi signal strengths received by the PDA from the various Access Points, and then corrected by filtering the signals (e.g. `KalmanFilter`) taking into account the route followed by the PDA.

The class `AdHocLocationService` inherits from `LocationService` and overrides two methods to define a more specific system behaviour adapted to the environment where the prototype is executed. Various classes (Contour in Figure 1) are used to display graphically the current state of the system. These display classes, as well as details about storage of data in files, the manipulation of Wi-Fi communication signals and position probabilities, and the classes that bind the other classes together, are implementation details that are abstracted in this view of the architecture.

2.3 Trust Requirements

Following the Trust Analysis Framework described in the deliverable WP2-01 [5], a new pervasive scenario was devised to illustrate the use of the prototype (by combining elements from the previous scenarios “A Pre-hospital Incident” and “A stay in a pervasive hospital environment”). The scenario highlighted the following trust issues:

- ⇨ Accuracy, with accuracy of the location estimate;
- ⇨ Audit trail, with the log files of the Web server;
- ⇨ Authorisation, with the centralised and restricted access to the shared resources;
- ⇨ Identification (Privacy), with confidentiality of exchanged information;
- ⇨ Availability, due to the possibility that the Wi-Fi signals do not reach all the PDAs;
- ⇨ Usability, which corresponds in this case to context-aware authorisation and is thus considered in other categories;
- ⇨ Harm (Privacy), with safety of wireless communications.

Several trust requirements of the prototype are not considered from an abstract point of view because they are dealt with at a low level or are not applicable:

- ⇨ Accuracy is solved via appropriate mathematical computations which estimate the position based on the physical communication signal strengths and then correct it, including how the signals from the wireless access points are represented; these elements of the system also contribute to the Reliability (integrity) issue;
- ⇨ Availability depends on the coverage of the set of Wireless Access Points and can be increased by adding some; in practice, the Wi-Fi bandwidth and signal coverage and largely enough;
- ⇨ Harm: current state-of-the-art research and legislation means that Wi-Fi is not an RF-safety issue.

We focus on requirements that are amenable to formal modelling and verification in this report. Usability aspects are thus not considered because they are not easily suited to formal modelling and verification.

The pervasive application we consider is not only adapted to the pervasive scenarios from WP2-01, but also relevant to many others devised by the pervasive community devised as it is the archetypal service driving context-aware applications. We have presented the general architecture of the pervasive application, what constitutes a first abstraction of the system.

Following the trust analysis of he underlying pervasive scenario, the location-based service has trust issues that can be reduced to Audit Trail, Authorisation and Identification (Privacy). We now address them while formally modelling the system in the next Section.

3 Formal Modelling

We describe the formal modelling activity that follows from the presentation of the pervasive application in the Section 2. The formal modelling language is briefly introduced in section 3.2, so that the various models of the pervasive application can be presented in sections 3.4 and 3.5.

3.1 Introduction

The computer programs which implement a software system are usually too detailed and complex to analyse the system at a sufficient level of abstraction for understanding its purpose and rigorously detect its faults. Modelling this system is necessary to be able to validate it against the set of desired abstract properties.

Formal methods are computing methodologies and techniques that stem from mathematics and, as such, are solid and powerful methods to produce accurate models of a system. This property of accuracy contrasts with the conceptual looseness of programming languages and forces the modeller to think about the system in a different way, what generally brings to clarification about design decisions and system properties. In the T-SAS project, we chose the B method as a formal method to model the pervasive application.

3.2 Brief Introduction to the B method

The B method is a formal method of software engineering. It is based on the B notation [1] and is supported by two commercial toolkits (Atelier B/B4FREE from ClearSy and the B-Toolkit from B-Core), with several other tools under development. It has been used to model and verify several industrial systems, for example in the railway industry, thus proving its applicability and usefulness to industrial systems.

The B notation is used to define Abstract Machines which have:

- ⇨ a state, made of **variables** and **constants** satisfying an **invariant**,
- ⇨ and various **operations**, that can modify the state, but must satisfy the invariant.

All data in B is modelled as simple mathematical entities (**sets**, **functions**, **sequences**) and the various logical formulas are expressed in predicate logic. Operations have **preconditions** and express desired postconditions via composition of basic **substitutions**. A B **machine** is composed of a sequence of **clauses**, typically of the following form:

```
MACHINE            machine_name
DEFINITIONS        syntactic_elements
SETS               set1={...}; set2={...}; set3; ...
CONSTANTS          constant1, constant2, ...
PROPERTIES         properties_of_the_constants
VARIABLES          variable1, variable2, ...
INVARIANT          invariant_over_variables
INITIALISATION     initial_state
OPERATIONS         operation1=…; operation2=…; ...
END
```

The `DEFINITIONS` clause defines a replacement text that is identified by a name and that can be referenced by its name in the B machine. The B notation offers several other features to structure a specification into several machines and refine a given (abstract) machine by adding details to it. The final **refinement** (called **implementation**) corresponds to an executable program. A different notation, called Event-B [2], has been introduced to model more appropriately event and distributed systems and introduces some restrictions and adds new features to the B method, but we will not discuss and use it in this report. A B machine is consistent if the initialisation and the operations preserve the invariant, and if the (possible)

refinement is correct. The semantics underlying the B notation defines the mathematical Proof Obligations (POs) that must be met to ensure these properties. In the final step of the B method, the POs must then be proven with the B toolkits, automatically for most of them, interactively for the most difficult.

The B notation is a modelling language and not a programming language (unless only implementations are considered), because it is used at a different level of abstraction. The following features show this difference:

- ⇨ Complex data structures can be represented more simply using abstract mathematical structures;
- ⇨ The definition of a variable or a constant can be deferred, or its value can be given by properties expressed in predicate logic;
- ⇨ The substitutions used in operations can be non-deterministic.

The following is an example of a B machine modelling a dictionary, which could correspond for example to a spellchecker program. This machine declares a set `WORD` representing the words that a dictionary can have; a constant natural number, `MaxSize`, which limits the size of the dictionary (the actual value is in the `PROPERTIES` clause); a variable `Words` corresponding to the set of words currently in the dictionary (the type information is in the `INVARIANT` clause); and three self-explanatory operations (only `checkWord` returns a value). This machine is consistent because one can verify that the operation preconditions (between the keywords `PRE` and `THEN`) ensure that the operation substitutions (between the keywords `THEN` and `END`) do not violate the invariant.

```
MACHINE                 Dictionary
SETS                    WORD
CONSTANTS               MaxSize
PROPERTIES              MaxSize: NAT1 & MaxSize=50
VARIABLES               Words
INVARIANT               Words: POW(WORD) & card(Words)<=MaxSize
INITIALISATION          Words:={}
OPERATIONS
            addWord(w) =
                PRE  w: (WORD-Words) & card(Words)<MaxSize
                THEN Words := Words \/ {w} END;
            removeWord(w) =
                PRE  w: Words
                THEN Words := Words - {w} END;
present <-- checkWord(w) =
                PRE  w: WORD
                THEN present := bool(w: Words) END
END
```

3.3 Assumptions made for the Modelling

The modelling of the pervasive application is first based on the architecture that was presented in Section 2.2, which already made some simplifications and abstractions. Accordingly, the model will ignore the low-level details of the application and concentrate on the abstract aspects so that trust properties can be appropriately modelled.

The model will more specifically focus on the user's point of view of the system, a necessary assumption in order to adequately reflect the pervasive nature of the system. For example, when choosing the operations of the B machines, we will give priority to those related to the user of the system and simplify, if possible, those related to the administration and configuration of the system.

The various models developed aim to formalise the relevant aspects of the pervasive application so that the properties of Audit Trail, Authorisation and Identification (Privacy) can be expressed (and formally verified, see Section 4). For that purpose, the concept of a policy is central to our models and corresponds to administrative information that is used by the

system to make a decision automatically. This concept in itself is the subject of a lot of research (not only in computer science but also in law and social sciences) and we will not discuss it specifically in this report.

Lastly, our models are constrained by the model-checking that we will apply later. Model-checking consists in the automatic exploration of the state-space of a specification and will be presented in more details in Section 4. The size and complexity of some elements of the B machines are limited so that model-checking is facilitated and, in some situations, exhaustive. This puts some limitation on the scope of the models, since they cannot model "large" systems, but the models are still useful at helping at designing the system and detecting faults.

3.4 First Case: Fixed Policy

We here describe the first B model, named `FixedPolicy`, where the policy was considered constant. The complete specification of the B machine can be found in Annex 6.1, while we explain here the principles and design decisions.

The `FixedPolicy` B machine is composed of two operations that model the authorisation decisions relative to the execution of a service offered by a pervasive device and relative to the registration of a device to the system. The trust properties of the model are that documents belong to one and only one pervasive device, all service executions should be authorised and an audit trail of the service calls is maintained.

3.4.1 Sets, Constants, Variables and Typing Invariant

The sets `DEVICE`, `SERVICE`, `DOCUMENT`, and `LOCATION` correspond respectively to the devices known to the system (in the sense that they can be registered), the services offered by the devices, the documents stored on the devices and used by the services, and the possible locations of the devices. The variables `ProvidedServices`, `Documents` and `Locations` (defined in the typing invariant `invariant1`) are functions that represent the associations between devices and the respective sets. The services `Display` and `Access` correspond to the location-based services that the pervasive application implements. The limited number of locations (three here) can be interpreted conceptually as rooms in a building.

The variable `RegisteredDevices` is a set of devices (elements of `DEVICE`) that corresponds to the devices registered to the system. Any device not yet registered is considered an "intruder".

An important element of our B models is the notion of `SERVICECALL` (SC) defined as the Cartesian product `DEVICE*SERVICE*DEVICE*DOCUMENT`. A tuple `(a |-> b |-> c |-> d)` from the set `SERVICECALL` models the fact that the caller device `a` requests the execution of the service `b` by the callee device `c` on a document `d`. The variable `ServiceCalls` is the actual set of requested service calls, i.e. the Audit Trail.

The model contains two constants: the `Policy` and `MAXNumServCalls`, the maximum number of service calls stored in the variable `ServiceCalls`. Their type and values are given in the `PROPERTIES` clause.

The set `ID` models the system users, which own the devices and want to use their services. The users are assigned roles from the set `ROLE` and the documents have (access) attributes from the set `ATTRIBUTE`. The variables `Owns`, `Roles` and `DocumentAttributes` are functions that respectively represent the associations between the users and their devices and roles, and between documents and their attributes.

3.4.2 Second Part of the Invariant

The second part of the invariant (definition `invariant2`) defines the meaningful constraints over the variables and constants that model the trust properties. It is the conjunction of two main conditions:

- ⇨ The documents are stored on one and only one device; this constraint on the variable `Documents` expresses in simple terms the privacy issue that a document is only stored on its owner's device (but it can be accessed by calling an `Access` service if the `Policy` enables it);
- ⇨ The elements of `ServiceCalls` are in `Policy`, i.e. the Authorisation issue.

3.4.3 Operations

We should start by saying that the operations of `FixedPolicy` have preconditions, but the preconditions do not correspond to an Identification issue. More precisely, B operation preconditions indicate that the given substitutions only apply when preconditions are met, while nothing is guaranteed otherwise. In other words, the call to an operation outside of its precondition is not modelled but may be possible in the pervasive application.

The operation `executeService` models the query by a caller to execute a service on a callee relative to a document. Its semantics is composed of a `SELECT` statement conditioned by four conjuncts:

- ⇨ The Audit Trail is not full;
- ⇨ The caller and callee are registered devices;
- ⇨ This service calls is authorised by the `Policy`;
- ⇨ The locations of the two devices are close (because this is a location-based service).

If the four conditions are met, the service call is "executed" by being added to the `ServiceCalls` set. This is an abstraction of the real execution of the service in the pervasive application, which will do something useful for the user, but the effect of a particular service are not modelled here. We assume that the offered services are not administrative (and thus do not change the machine variables). If one of the four conditions is not met, the substitution is `skip`, which does nothing (once more, the pervasive application will probably do something in that case, like record the request to unauthorised service calls or create a new audit trail, but it is not modelled here).

The `registerDevice` operation models in simple terms the registration of a new device to the pervasive system. The only condition that the operation checks is that the person asking for the registration is a doctor, not a patient.

3.4.4 Initialisation

The prototype starts with a certain knowledge of the system, what corresponds to the initialisation of the B machine. For example, the identity of the PDAs and their owners are known to the system, as exemplified by the fact that the `Policy` uses the PDA identities (textual names, but it could be more complex, e.g. a cryptographic public key). Furthermore, the initial values of the prototype's variables can be given here, like the audit trail `ServiceCalls` that is initially empty.

The other initialisations correspond to environment of the prototype when it starts: where the documents are, what their attributes are; who owns what device, where it is located and what services it offers; and what are the roles of users.

3.5 Second Case: Changing Policy

Here, most of the elements of the model remain unchanged from the first case, but we modify the model so that `Policy` can be changed: it is a variable and no longer a constant. This modification corresponds to an important property of dynamic environments like Pervasive Computing, where devices come and go and user's privileges are passed to its new devices

automatically, so that the user's attention remain on its meaningful tasks and is not focused on system administration tasks.

Complete specifications of the B machine `ChangingPolicy` can be found in Annex 6.2.

3.5.1 Data and operation changes to the previous model

The various SETS definitions are unchanged, apart from the introduction of a role `Administrator`. `Policy` is a new variable, and `ServiceCalls` is divided into two sets: `RequestedSC` for the service calls that have been requested, and `GrantedSC` for those who have been granted (the structure of SC remains unchanged here). The first part of the invariant `invariant1` is modified accordingly.

The operation `executeService` is split into `grantService` and `denyService`, similarly to the SC variables. Those two methods have very similar bodies to `executeService` (`denyService` being the "negative" form of `grantService`, except that only the `RequestedSC` set is updated).

In addition to `registerDevice`, the `ChangingPolicy` machine specifies an `unregisterDevice` operation to unregister a device from the system. The definition of these methods is very different from those in `FixedPolicy`, because they express the additional feature that device (un)registration modify the `Policy`. The `Policy` is updated so that SC authorised for the other devices of the device owner are also authorised for the new device (and then removed after unregistration).

3.5.2 Invariant

Due to the changes to the B machine, the second part of the invariant `invariant2` becomes more complex. The first condition relative to documents remains unchanged, while the second condition relative to the size of `Policy` is removed (since the `Policy` may change a lot, this is no longer relevant).

The third condition is split into two conditions: `GrantedSC` must be a subset of `RequestedSC` (granted service calls must have been requested, this corresponds to the non-repudiation property) and of `Policy` (Authorisation issue).

A fourth condition is added to model an Identification issue: it states that only registered devices can have authorised (in `Policy`) or granted (in `GrantedSC`) service calls. This property is expressed as a logical formula ranging over all the SC (the symbol "`!`" corresponds to the logical operator `!`). One can notice that the second part of this condition and the fact that `GrantedSC` is a subset of `Policy` (previous condition) imply the first part, nevertheless the condition is left for the sake of clarity.

A fifth condition clarifies the privacy property by saying that if a service call is granted, then a call to the `Access` service from the same caller to the same callee regarding the same document should be authorised. Since the set `SERVICE` has only two elements, service to be granted, the access to the document should be authorised if a `Display` service is to be granted.

3.5.3 Initialisation

While the style of initialisation used in `FixedPolicy` is still valid, we chose to use another style of initialisation here. The B notation enables to define the value of a variable by "choice by predicate":
`S: p` (where `S` is a variable and `p` a predicate, i.e. a valid logical formula) specifies that `S` should take any value satisfying the condition `p`.

The reason why we specify the machine initialisation this way will be explained in Section 4.3.2, but roughly corresponds to the fact that we are using here more complex structures for the policy (see Section 3.5.4) and wish to test how they behave.

3.5.4 Different models of the Policy

The main activity regarding the modelling of this B machine consisted in experimenting with different representation of the `Policy`. As shown above, we first started with tuples of size four (caller,service,callee,document), as for the `FixedPolicy` machine. Although this structure is simple and intuitive, treatments of its elements can be quite complex as the bodies of the operations `registerDevice` and `unregisterDevice` show.

We experimented with another way to structure the `Policy` in order to simplify the writing the operations. In this other B machine, `Policy` was defined (in the second part of the `INVARIANT`) as:

```
Policy: DEVICE --> SERVICECALL
```

With `SERVICECALL == SERVICE*DEVICE*DOCUMENT`. This way, the operation `unregisterDevice` was reduced to a simple domain subtraction:

```
Policy := Policy <<| {device}
```

But on the other hand, the body of the operation `registerDevice` was made more complex by the use of a lambda expression (not detailed here) to define the function to update `Policy` by its properties.

3.6 Conclusion

We showed that several models of the pervasive application can be created using the B method and depending on the various assumptions made. The simple `FixedPolicy` machine proved useful at first getting an idea of the modelling task and difficulties. The more complex and complete `ChangingPolicy` enabled us to model more precisely the application properties and experiment with the policies.

The principal observation that resulted from the modelling activity is that, after a certain level of model complexity, data structure is greatly influenced by the choice of operations that are deemed important (either because they are used frequently or because they are critical to the system functioning). More interestingly, the structure of the `Policy` depends on the operations that will frequently modify it and reversely makes more complex some other operations.

To conclude, the modelling activity is an ongoing task, which helps clarifying the various aspects of the system (properties, design, architecture, etc.) at every step. It is important to start modelling at a sufficiently high level of abstraction in order to be able to model trust properties, which are holistic in that they apply to many scattered elements of the system. Further steps of the modelling will lead us to refine the abstract models until we reach a specification corresponding to the prototype code.

4 Formal Validation and Verification

We describe the formal verification and validation that follows from the modelling in Section 3. The formal verification and validation tool is presented in Section 3.2. Basic assumptions are presented in Section 3.3 and then the two modelling cases are considered in Sections 3.4 and 3.5.

4.1 Introduction

The modelling task can bring insight into the flaws and the difficulties relative to the design of a system, but it is not sufficient to discover detailed errors. It is essential to spend time and efforts verifying that the model is correct, and try to discover errors that will reveal a design fault or an unexpected behaviour.

Many techniques exist for the validation and (formal) verification of models, including testing, theorem proving and model-checking. They are all supported by tools to automate the verification task. In the case of the B method, rather than using the B toolkits to mathematically prove our B machines, we chose to use the ProB model-checker, which automatically explores the state-space of a B machines. It automates the verification of several checks on the B machines and provides an animator for B machines so that they can be validated against a given scenario.

4.2 Short Presentation of ProB

ProB [4] is a model-checker which attempts to automatically find inconsistencies of B machines. Figure 2 shows a snapshot of the tool. It applies to the B machine a certain number of basic checks (syntax, types), then tries to find a set of valid initialisations and finally gives the choice to the verifier to either animate the model, or model-check it.

Animation corresponds to the process of displaying the effects of B machine operations on its state. Usually implemented via an interactive visual interface, it is a method that enables a designer to test its models against a given scenario (i.e. sequence of operations) and the expected final state, in that way validating the model. ProB automatically computes at each step of the animation a set of enabled operations (i.e. such that their parameters meet the preconditions), with values for their parameters, so that the modeller only have to click on the desired element of the list of enabled operations. ProB also proposes a BACKTRACK operation to go back to the previous step in the animation, and can animate a B machine by randomly selecting operations.

Model-checking of a B machine is the exploration of its state-space, by enumerating possible values for the various undetermined elements (deferred sets and variables, parameter values for the operations). In fact, ProB offers the option to manipulate certain elements symbolically, i.e. working on the expression of these elements rather than on their enumerated values. ProB has two modes of model-checking:

- ⇨ Temporal model-checking, to look for sequences of operations starting from a valid initial state and leading to a violation of the invariant (or other situations, like deadlock and abort when no operations are enabled due to the impossibility to find values satisfying respectively the guards and the preconditions for their parameters),
- ⇨ Constraint-based model-checking, to look for a state satisfying the invariant where a single operation can be applied to reach a state where the invariant is violated.

ProB represents the explored state-space by a Labelled-Transition System (LTS) whose nodes are labelled by the state (values of constants and variables) and whose transitions are labelled by an operation name (which corresponds to the application of the operation). ProB offers a graphical visualization of the state-space (see Annex 6) and of the sequence of

operations leading to the current state. Many aspects of the ProB tool can be configured via a preference menu to adapt the animation, or model-checking, to a particular need.

The coverage of the model-check depends on the size of the sets and the number, and complexity, of the operations. The model-check can be exhaustive if these numbers are small, thus proving that the B machine is consistent. But, contrarily to proving the POs with the B toolkits, non-exhaustive model-checking can generally only show the presence of errors, not their absence.

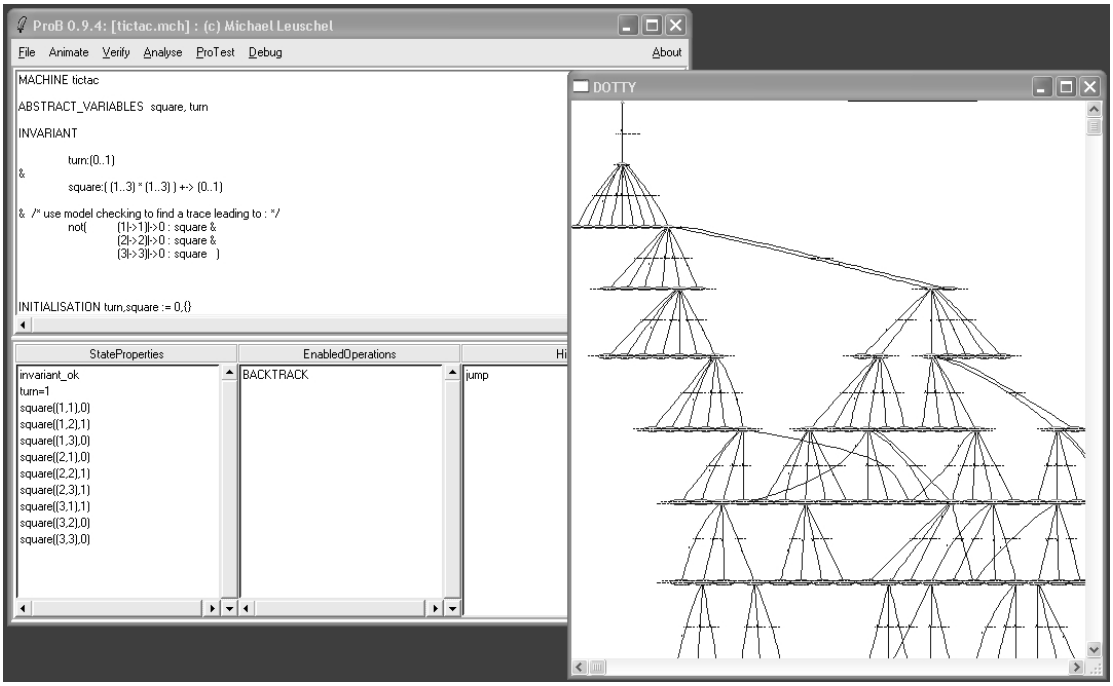

Figure 2

4.3 Results of the Formal Verification

4.3.1 Constraints imposed on the Models

Though modelling and model-checking are activities that can work independently, the models must, in practice, often be adapted to the model-checking technique and tools in order to make those later efficient, because the coverage rate depends on elements of the models. In the case of B and ProB, the sets have to be limited in size and the operations in complexity, so that the state-space is not too big.

In the examples of Section 3, positions are modelled by a very small number of locations. This implies that the condition of device locations proximity is overly simplified by an equality comparison, while the pervasive application computes the Euclidian distance between the two devices. This particular case seems, nevertheless, a reasonable abstraction.

We can also notice that in the second case, the sets are even more limited in size than in the first case (two devices, IDs and locations, and one document) because there are more variables and operations.

4.3.2 First Case

This simple model was easy to model-check, because the set sizes and operation complexity were low. Different versions of the B machine were checked depending on:

- ⇨ The initialisation of `RegisteredDevices`;
- ⇨ The presence of the test `Locations(callee)=Locations(caller)`.

In all the cases, the B machines were consistent.

The most difficult situation to check is when RegisteredDevices is initialised to the empty set {} and the test is present. The state-space size is then 15 (see Figure 3 in Annex 6.1.2). The state-space is reduced if devices are added to RegisteredDevices and if the test is removed, for example Figure 4 (Annex 6.1.2) shows the situation of an empty initialisation and no test with a state-space of size 8. We can notice on the two figures in Annex 6.1.2 that the state-spaces have a similar structure, what corresponds to the fact that the behaviours are the same.

The reason why the state-space is small is that ServiceCalls remains empty, which itself is due to the fact that the Policy only authorises two service calls on different services and that executeService requires the locations to be the same. This in turn comes from the fact that the three devices are in three different locations, but no operation changes the locations. We modified this behaviour by adding a changeLocation operation to the B machine:

```
changeLocation(device,location)=
PRE
      device: DEVICE & location: LOCATION
THEN
      Locations(device) := location
END
```

The corresponding B machine has a state-space of size approximately 350 and is still proven consistent via an exhaustive model-checking with ProB.

4.3.3 Second Case

Animation and model-checking of the ChangingPolicy B machines was more interesting than for the FixedPolicy machine as it brought up several situations that lead us to correct (for example, to ensure the privacy property in operations grantService and denyService, or to initialise the B machine with Access services available so that privacy properties can be met) and modify the models. Furthermore, some powerful features of ProB enabled the exploration of the behaviour of those B machines.

For example, as shown in Section 3.5.3, the initialisation of the ChangingPolicy is written with a choice by predicate as follows:

```
      Policy, RegisteredDevices, Documents, Owns, Location, Roles,
      ProvidedServices, DocumentAttributes, RequestedSC, GrantedSC
            :(invariant1 & invariant2 &
              Documents       = {Pda1|->{Xray1}, Pda2|->{}} & ...)
```

This initialisation states that all the initial values of the variables should satisfy the invariant and some explicit assignments (if those assignments are incompatible with the invariant, the B machine is inconsistent). When launching the B machine, ProB tries to enumerate the valid assignments for all the the variables, which can then be used by the modeller to animate the machine or which can be followed further during model-checking.

The various ChangingPolicy B machines contained inconsistencies detected by ProB. Most of the problem discovered came from invariant violations and the ability of ProB to display the truthfulness of the various conditions composing the invariant allowed to confine quickly and accurately the problem. Very interesting situations occurred, like for example an invariant violation occurring after a sequence of 8 operations (state-space size around 350, see Annex 6.3) which originated from a mistake in the writing of the (un)register operations.

Finally, invariant violations were found (see the red nodes in Figure 5 and Figure 6) and pointed at some pertinent problems. For example, after a device unregistration, some “dead” authorisation tuples would stay in Policy (violating the condition that tuples in the Policy should only have registered devices) because of the way Policy was updated by considering only the first element of each tuples (the caller, see Policy modelled as a function in Section 3.5.4).

Some other detected invariant violations were caused by a more general problem. This problem is that `GrantedSC` would no longer be a subset of `Policy` due to the updates of this `Policy` after unregistration of devices, thus invalidating the condition "`GrantedSC :> Policy`" of the invariant. This meant that this later condition was too strong and that the model of `Policy` was incomplete to ensure some trust properties (Authorisation in the above example). More generally, it expresses the idea that variables evolving during the system execution should be considered by group of "dynamic properties", in the sense that modifications on one group during execution should be reflected by equivalent modifications on the other groups. Applied to the above example, `Policy` could be completed by a set `Unregistered` recording the unregistration of devices, so that the two groups (`Policy`, `Unregistered`) and (`GrantedSC`) can be kept consistent.

4.4 Conclusion

We showed how we completed our modelling activity by the formal verification of the created models. This proved successful in that the ProB model-checker performed well and was useful in finding problems in the models. The simple B machine `FixedPolicy` was easily proven correct, but the more interesting situations occurred with the `ChangingPolicy` B machines. ProB helped in the understanding of the models via animation and in their correction via model-checking.

5 Conclusions

5.1 Summary

In conclusion, we have clarified the trust properties highlighted in the previous deliverable by modelling and verifying formally a pervasive application inspired from the previous pervasive scenarios. We have shown that the modelling task, using the B method, and the verification task, using the ProB model-checker, work hand in hand very well. They enabled us to discover some interesting behaviours, which lead us to solve non-obvious problems, and progress step by step towards an accurate model of the pervasive application. But, as said previously, modelling is an ongoing activity that requires a lot of maturation.

Finally, the fact that only one particular pervasive application is considered in this deliverable does not limit the scope of the discovered results (similarly to the fact that the pervasive scenarios developed in the deliverable WP2-01 did not limit the scope of Trust Analysis Grid created on top of them). We concretise this by giving some guidelines in the form of a methodology in the next Section.

5.2 Guidelines

As a conclusion, we would like to highlight the methodology we used during our modelling and verification activities. This methodology summarises several of the results given in this report. It could be helpful to guide users of formal method and tools during their first attempts to formally prove the (trust) properties of their system. The methodology follows and does not assume any particular formal method or tool:

1) Model important features of the system
   First vaguely type the variables; then write a set of operations corresponding to complementary features while (possibly) modifying the variable types to ease this writing; consider the variables by group of similar dynamic properties;
2a) Model-check the model
   2a.a) Property violation detected
      Examine the various aspects of the model (variables, enabled operations, history of operations) to see what part of the property is “false”; Correct the model accordingly;
   2a.b) No property violation detected
      Go back to 2a until coverage rate is enough; possible changes to the model include: modify the initialisation to test other situations (in B use “choice by predicate”); add inconsistencies in the model;
2b) Animate the model
   2b.a) Execute the desired sequence of operations (validation);
   2b.b) Find an interesting state, then 2a.a or 2a.b applies;
   2b.c) Backtrack from a state where the invariant is violated;
3) Go back to 1 (complete the model) or refine the model.

5.3 Future work

As we said, more modelling of the pervasive application is needed to solidify our results and translate them back to concrete proposals for the prototype. Regarding that aspect, we would like to use UML diagrams in order to ease the presentation of the formal models to the prototype developers. In particular, we could use the UML-B Profile [6] to automate the process of translation between B and UML. Refinements of our models will provide interesting supports for discussion between the modellers and the development team.

As this report highlighted, the concept of policy is central and is currently the subject of a lot of research. A possible direction of work would be the study and use of (Trust) Policy Languages that could be concretely applied in the case of the pervasive application.

Acknowledgements

The authors would like to acknowledge the help of Andrew Edmunds in the creation and correction of the B models.

Bibliographic references

[1] J.-R. Abrial, *The B-Book*, Cambridge University Press, 1996.

[2] J.-R. Abrial, *Extending B without changing it (for developing distributed systems)*, in H. Habrias, ed., *1st Conference on the B method*, 1996, pp. 169-190.

[3] C. Jensen, S. Poslad and T. Dimitrakos, eds., *Trust Management, Second International Conference on Trust Management, iTrust 2004*, Springer, Oxford, UK, 2004.

[4] M. Leuschel and M. Butler, *ProB: A Model Checker for B*, in S. G. A. Keijiro, M. Dino, ed., *Formal Methods Europe*, Springer, Pisa, Italy, 2003, pp. 855-874.

[5] S. L. Presti, M. Cusack, C. Booth, D. Allsopp, M. Kirton, N. Exon, P. Beautement, M. Butler, M. Leuschel and Phillip Turner, *Trust Issues in Pervasive Environments, Deliverable WP2-01 (version 1.1)*, http://www.trustedagents.co.uk/TSA-WP2-01v1.1.pdf, 2003.

[6] C. Snook and M. Butler, *Towards a UML profile for UML-B*, University of Southampton, UK, 2003.

6 Annex 1
B machines and their State-spaces

6.1 Fixed Policy Model

6.1.1 `FixedPolicy` B machine

```
MACHINE FixedPolicy
SETS
  DEVICE   = {Pda1, Pda2, Pda3};
  SERVICE  = {Display, Access};
  DOCUMENT = {Xray1, Xray2};
  ID       = {Alice, Bob, Charlie};
  ROLE     = {Doctor, Patient};
  LOCATION = {P1, P2, P3};
  ATTRIBUTE = {Read, Write}

CONSTANTS
  Policy, MAXNumServCalls

DEFINITIONS
  /* ServiceCall correspond to tuples (caller,service,callee,document) */
  SERVICECALL==DEVICE*SERVICE*DEVICE*DOCUMENT;
/* invariant defining the type of variables */
invariant1 == (
  RegisteredDevices:   POW(DEVICE) &
  Documents:           DEVICE --> POW(DOCUMENT) &
  Owns:                ID --> POW(DEVICE) &
  Locations:           DEVICE --> LOCATION &
  Roles:               ID --> POW(ROLE) &
  ProvidedServices:    DEVICE --> POW(SERVICE) &
  DocumentAttributes:  DOCUMENT --> POW(ATTRIBUTE) &
  ServiceCalls:        POW(SERVICECALL)
    );
/* second part of the invariant with the interesting properties */
invariant2 == (
  /* documents are not shared */
  !(a1,a2).(a1: dom(Documents) & a2: dom(Documents)
          & a1/=a2 => Documents(a1)/\Documents(a2)={}) &
  union(ran(Documents)) = DOCUMENT &
  /* limit on the audit trail size */
  card(ServiceCalls) <= MAXNumServCalls &
  /* Authorisation: all ServiceCalls should be authorised by the Policy */
  ServiceCalls <: Policy)

PROPERTIES
/* PROPERTIES defining the CONSTANTS */
  Policy: POW(SERVICECALL) &
  Policy = {Pda1 |-> Display |-> Pda2 |-> Xray1,
            Pda2 |-> Access |-> Pda3 |-> Xray2} &
  MAXNumServCalls: NAT & MAXNumServCalls = 10

VARIABLES
  RegisteredDevices, Documents, Owns, Locations, Roles,
  ProvidedServices, DocumentAttributes, ServiceCalls
```

```
INVARIANT
/* INVARIANT on the VARIABLES */
  invariant1 & invariant2

INITIALISATION
/* INITIALISATION of the VARIABLES */
  RegisteredDevices    := {} ||
  Documents            := {Pda1|->{Xray1}, Pda2|->{}, Pda3|->{Xray2}} ||
  Owns                 := {Alice|->{Pda1}, Bob|->{Pda2}, Charlie|->{ Pda3}} ||
  Locations            := {Pda1|->P1, Pda2|->P2, Pda3 |->P3} ||
  Roles                := {Alice|->{Doctor}, Bob|->{Doctor},
                           Charlie|->{Patient}} ||
  ProvidedServices     := {Pda1|->{}, Pda2|->{Display},
                           Pda3|->{Display,Access}} ||
  DocumentAttributes   := {Xray1|->{Read}, Xray2|->{Read,Write}} ||
  ServiceCalls         := {}

OPERATIONS
/* the operation executeService models the attempt to execute a service */
executeService(caller,service,callee,document) =
PRE
  caller: DEVICE & callee: DEVICE & service: SERVICE & document: DOCUMENT &
  service: ProvidedServices(callee)
THEN
  SELECT card(ServiceCalls) < MAXNumServCalls &
    caller: RegisteredDevices & callee: RegisteredDevices &
    caller |-> service |-> callee |-> document: Policy &
    Locations(caller) = Locations(callee) THEN
      /* Audit Trail: kept track of the service call */
      ServiceCalls := ServiceCalls \/ {(caller|->service|->callee|->document)}
END;

/* the operation registerDevice models the attempt to register a new device in
     the system */
registerDevice(person,device)=
PRE
  person: ID & device: DEVICE
THEN
  IF Doctor: Roles(person) THEN
    /* Identification (Privacy): only doctors KNOW the process to register a
     device */
    RegisteredDevices := RegisteredDevices \/ {device}
  ELSE
    /* Authorisation: unauthorised registration */
    skip
  END
END
END
```

6.1.2 `FixedPolicy` State-space

In this graph (and all the others), the loops (transition to the same state) have been removed. Figure 3 corresponds to `FixedPolicy` with initialisation of `RegisteredDevices` to empty set and test on locations in `executeService`.

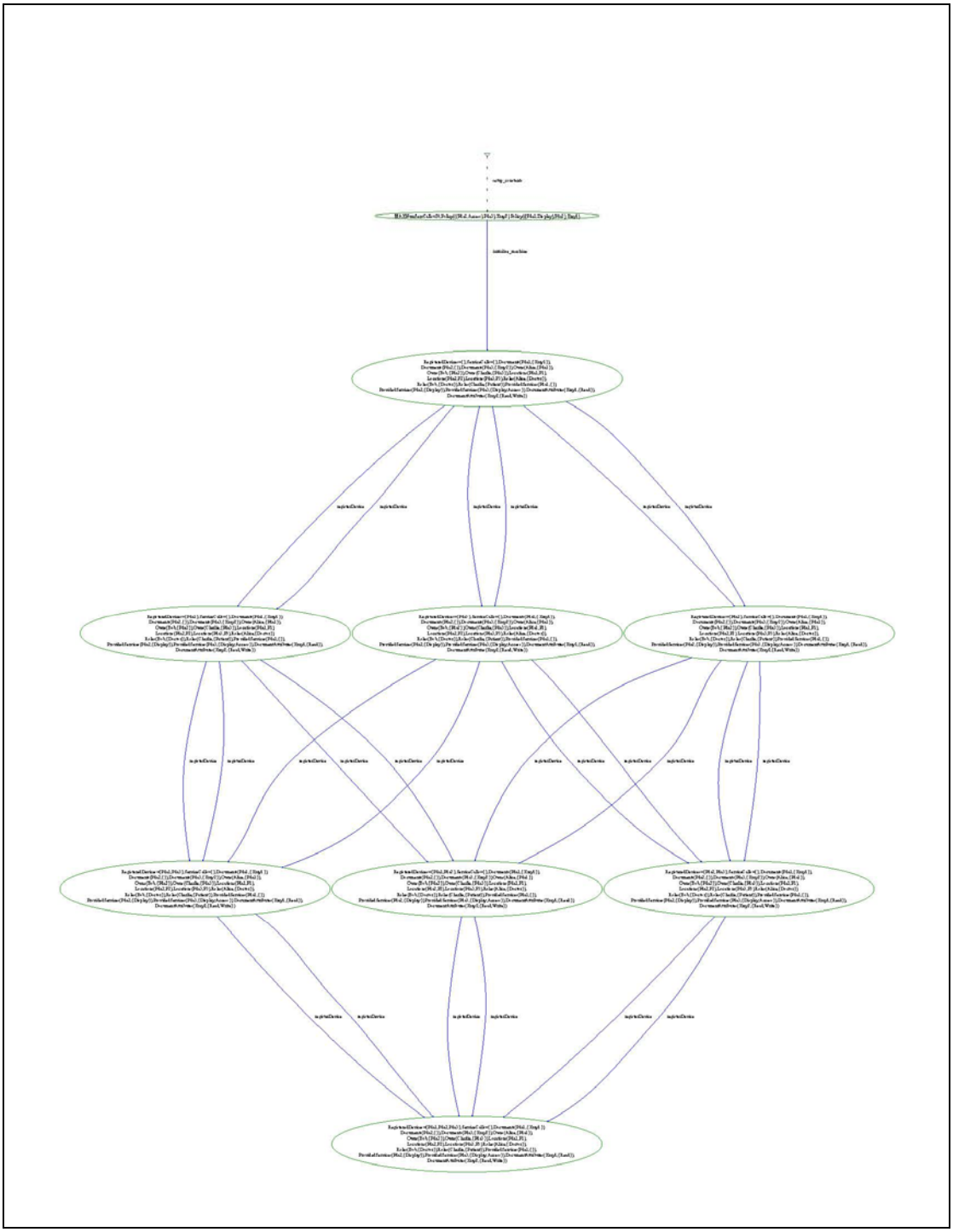

Figure 3

Figure 4 corresponds to the B machine where the test on `Locations` is removed:

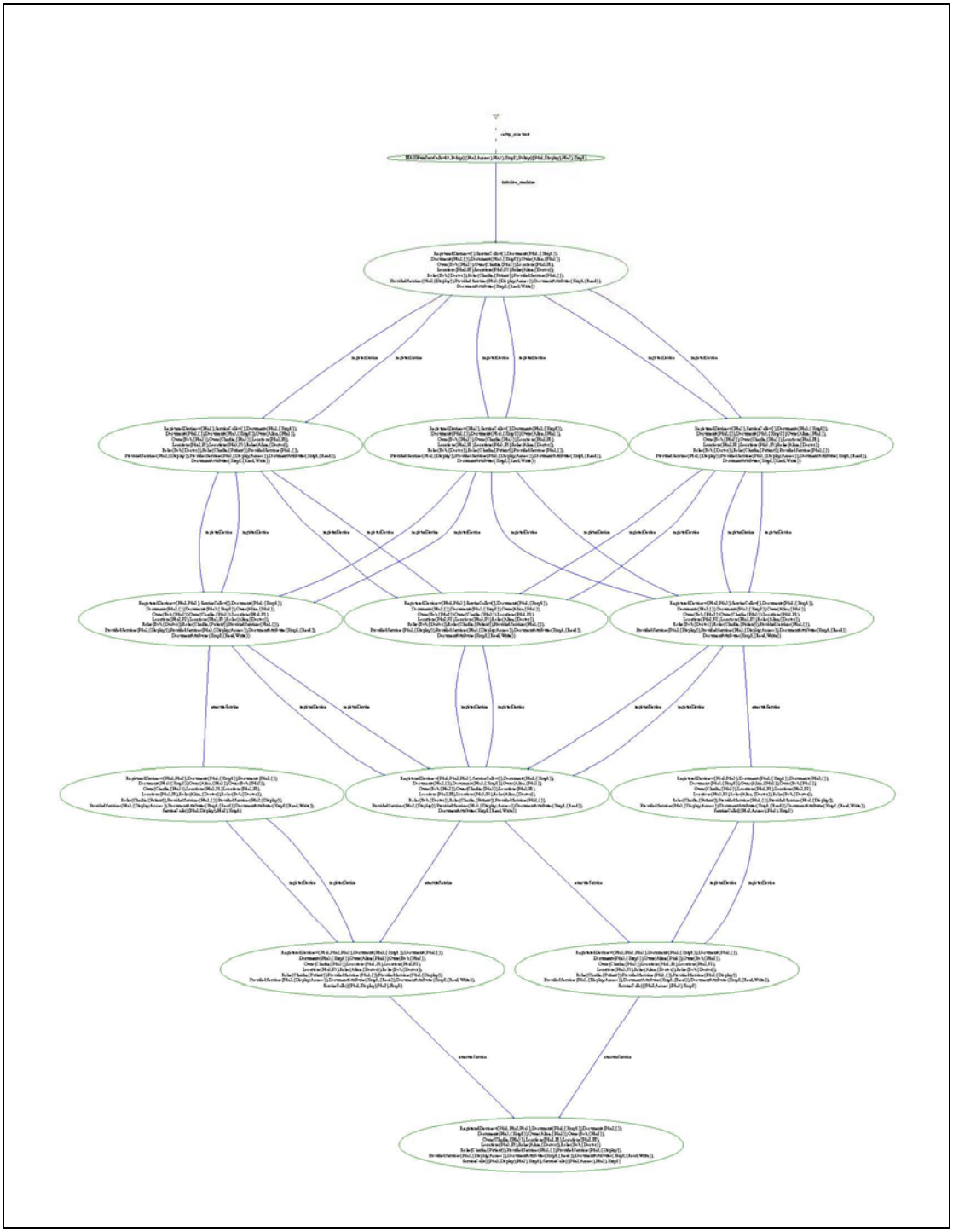

Figure 4

6.2 Changing Policy Models

6.2.1 ChangingPolicy B machine

```
MACHINE ChangingPolicy
SETS
     DEVICE       = {Pda1, Pda2};
     SERVICE      = {Display, Access};
     DOCUMENT     = {Xray1};
     ID           = {Chris, Michael};
     ROLE         = {Doctor, Administrator};
```

```
    LOCATION    = {P1, P2};
    ATTRIBUTE   = {Read, Write}

DEFINITIONS
    /* ServiceCall: (caller,service,callee,document) */
    SERVICECALL==DEVICE*SERVICE*DEVICE*DOCUMENT;
/* invariant1: typing invariant */
invariant1 == (
    Policy:           POW(SERVICECALL) &
    RegisteredDevices: POW(DEVICE) &
    Documents:        DEVICE --> POW(DOCUMENT) &
    Owns:             ID --> POW(DEVICE) &
    Location:         DEVICE --> LOCATION &
    Roles:            ID --> POW(ROLE) &
    ProvidedServices: DEVICE --> POW(SERVICE) &
    DocumentAttributes: DOCUMENT --> POW(ATTRIBUTE) &
    RequestedSC:      POW(SERVICECALL) &
    GrantedSC:        POW(SERVICECALL)
);
invariant2 == (
    /* documents are not shared between devices and they are all are stored on
       a device */
    !(a1,a2).(a1: dom(Documents) & a2: dom(Documents)
          & a1/=a2 => Documents(a1)/\Documents(a2)={}) &
    union(ran(Documents)) = DOCUMENT &
    /* SC granted must have been requested and be in the policy */
    GrantedSC <: RequestedSC &
    GrantedSC <: Policy &
    /* identification: only registered devices can be in the policy or be
      granted authorisations */
    !(a,b,c,d).(a:DEVICE & b: SERVICE & c: DEVICE & d: DOCUMENT &
          (a|->b|->c|->d):GrantedSC => a:RegisteredDevices
           & c:RegisteredDevices & a/=c) &
    !(a,b,c,d).(a:DEVICE & b: SERVICE & c: DEVICE & d: DOCUMENT &
          (a|->b|->c|->d): Policy => a /= c & a:RegisteredDevices) &
    /* privacy: for a service to be called, the device should have "Access" to
       the document */
    !(a,b,c,d).(a:DEVICE & b: SERVICE & c: DEVICE & d: DOCUMENT &
          (a|->b|->c|->d): GrantedSC => (a|->Access|->c|->d): Policy)
)

VARIABLES
    Policy, RegisteredDevices, Documents, Owns, Location, Roles,
    ProvidedServices, DocumentAttributes, RequestedSC, GrantedSC

INVARIANT
    invariant1 & invariant2

INITIALISATION
Policy, RegisteredDevices, Documents, Owns, Location, Roles,
ProvidedServices, DocumentAttributes, RequestedSC, GrantedSC
    :(invariant1 & invariant2 &
      Documents       = {Pda1|->{Xray1}, Pda2|->{}} &
      Owns            = {Chris|->{Pda1}, Michael|->{Pda2}} &
      Location        = {Pda1|->P1, Pda2|->P1} &
      Roles           = {Chris|->{Doctor,Administrator}, Michael|->{Doctor}} &
      ProvidedServices = {Pda1|->{}, Pda2|->{Display}} &
      DocumentAttributes = {Xray1|->{Read}} &
      RequestedSC     = {}  & GrantedSC = {})

OPERATIONS
```

```
grantService(caller,service,callee,document) =
PRE
  caller: DEVICE & callee: DEVICE & service: SERVICE & document: DOCUMENT
THEN
/* Authorisation mechanism, decomposed into: */
  /* devices must be registered AND service offered by callee */
   SELECT (caller: RegisteredDevices & callee: RegisteredDevices &
                 service: ProvidedServices(callee)
  /* service call must be authorised */
  & (caller |-> service |-> callee |-> document): Policy
  /* devices not close enough */
  & Location(caller) = Location(callee)
  /* to ensure privacy */
  & (service /= Access) => (caller|->Access|->callee|->document): Policy)
  THEN
    RequestedSC := RequestedSC \/ {caller|->service|->callee|->document}  ||
    GrantedSC := GrantedSC \/ {caller|->service|->callee|->document}
  END
END;

denyService(caller,service,callee,document) =
PRE
     caller: DEVICE & callee: DEVICE & service: SERVICE & document: DOCUMENT
THEN
  /* see grantService operation for details about the guard */
  SELECT not (caller: RegisteredDevices & callee: RegisteredDevices &
           service: ProvidedServices(callee)
  & (caller |-> service |-> callee |-> document): Policy
  & Location(caller) = Location(callee)
  & (service /= Access) => (caller|->Access|->callee|->document): Policy)
  THEN
    RequestedSC := RequestedSC \/ {caller|->service|->callee|->document}
  END
END;

registerDevice(person,device)=
PRE
  person: ID & device: DEVICE
THEN
  SELECT device: Owns(person) & Administrator: Roles(person) THEN
    /* only administrators KNOW the process to register a device */
    RegisteredDevices := RegisteredDevices \/ {device} ||
    /* all the authorisations of the person are granted for the new device */
    Policy := Policy \/ {varx| varx: SERVICECALL &
      #(aa,bb,cc,dd).(aa: DEVICE & aa: Owns(person) & bb: SERVICE &
                      cc: DEVICE & dd: DOCUMENT & varx=(device|->bb|->cc|->dd) &
                     (aa|->bb|->cc|->dd): Policy)}
    END
END;

unregisterDevice(person,device)=
PRE
  person: ID & device: RegisteredDevices
THEN
  SELECT device: Owns(person) & Administrator: Roles(person) THEN
    RegisteredDevices := RegisteredDevices - {device} ||
    /* all the authorisations granted for the device are removed */
    Policy := Policy - {varx| varx: SERVICECALL &
      #(aa,bb,cc,dd).(aa: DEVICE & aa: Owns(person) & bb: SERVICE &
                      cc: DEVICE & dd: DOCUMENT & varx=(device|->bb|->cc|->dd) &
                     (aa|->bb|->cc|->dd): Policy)}
```

```
   END
END
END
```

6.2.2 `ChangingPolicy` State-spaces

In the following graph, node labels are simplified, red nodes correspond to invariant violation and transition labels are removed. Figure 5 represents the model-checking of `ChangingPolicy` with the default ProB exploration:

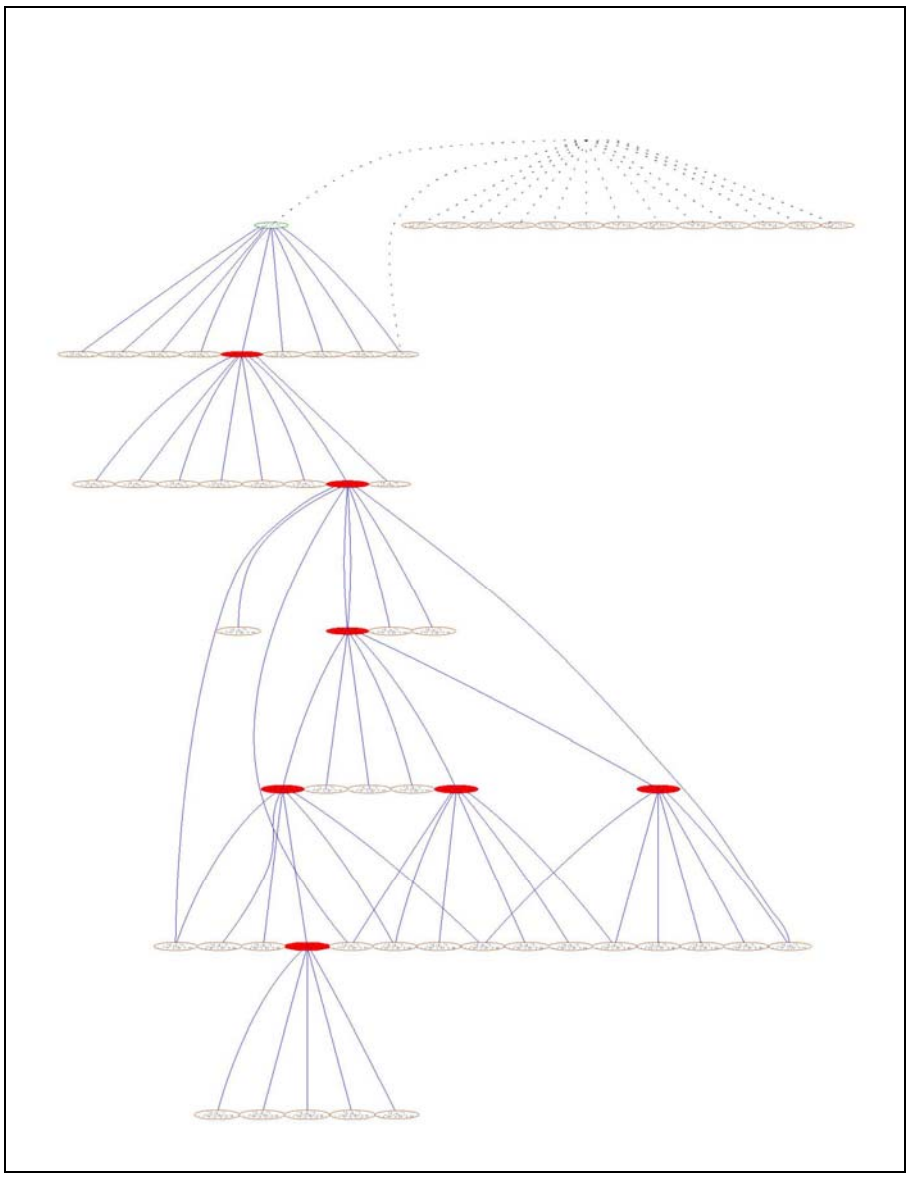

Figure 5

Figure 6 correspond to the model-checking of the same B machine but with breadth-first exploration of the state-space:

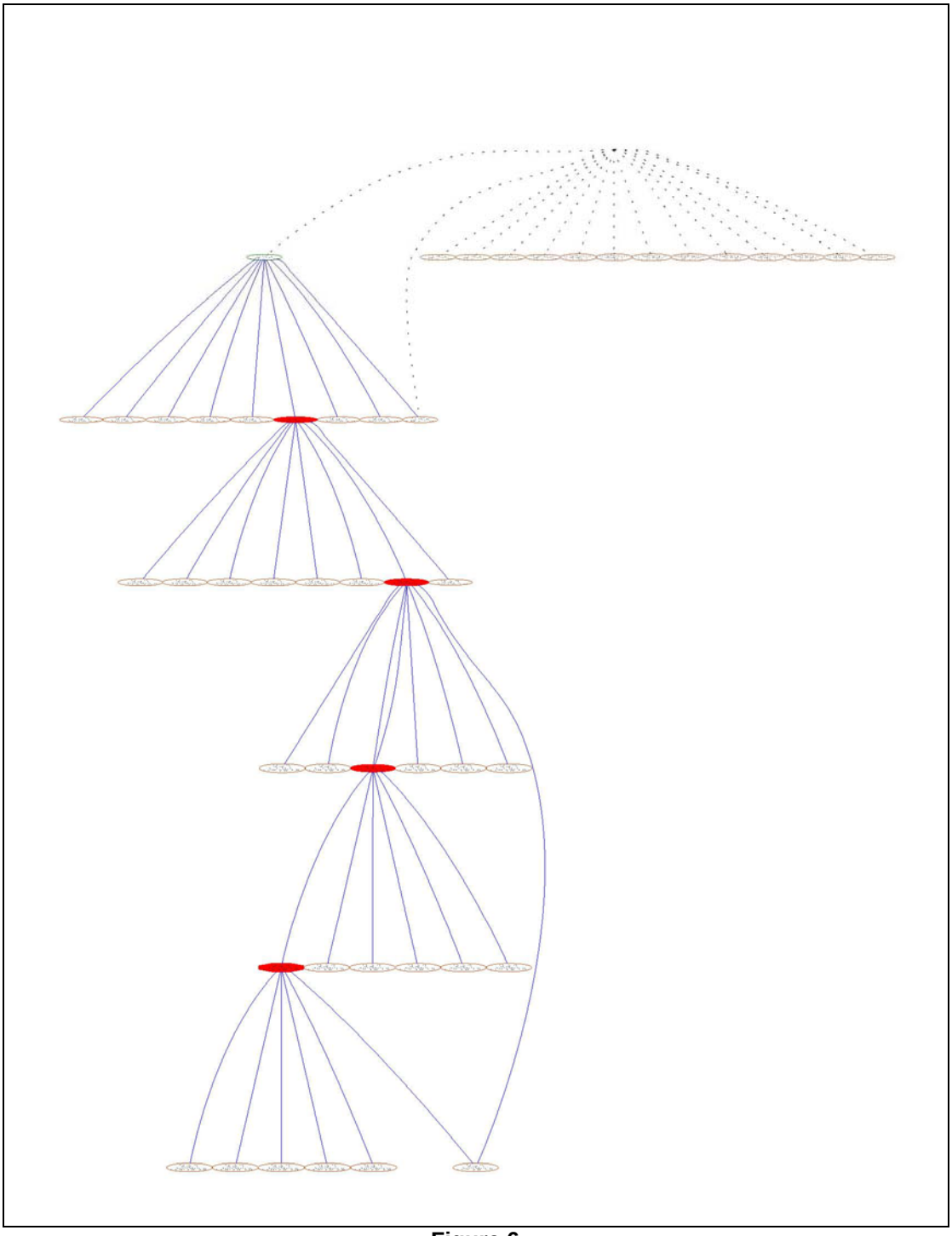

Figure 6

6.3 Visualisation of a large B machine state-space

Figure 7 represents a state-space of size approximately 350, with 8 levels of operation applications. One of the bottom nodes correspond to an invariant violation.

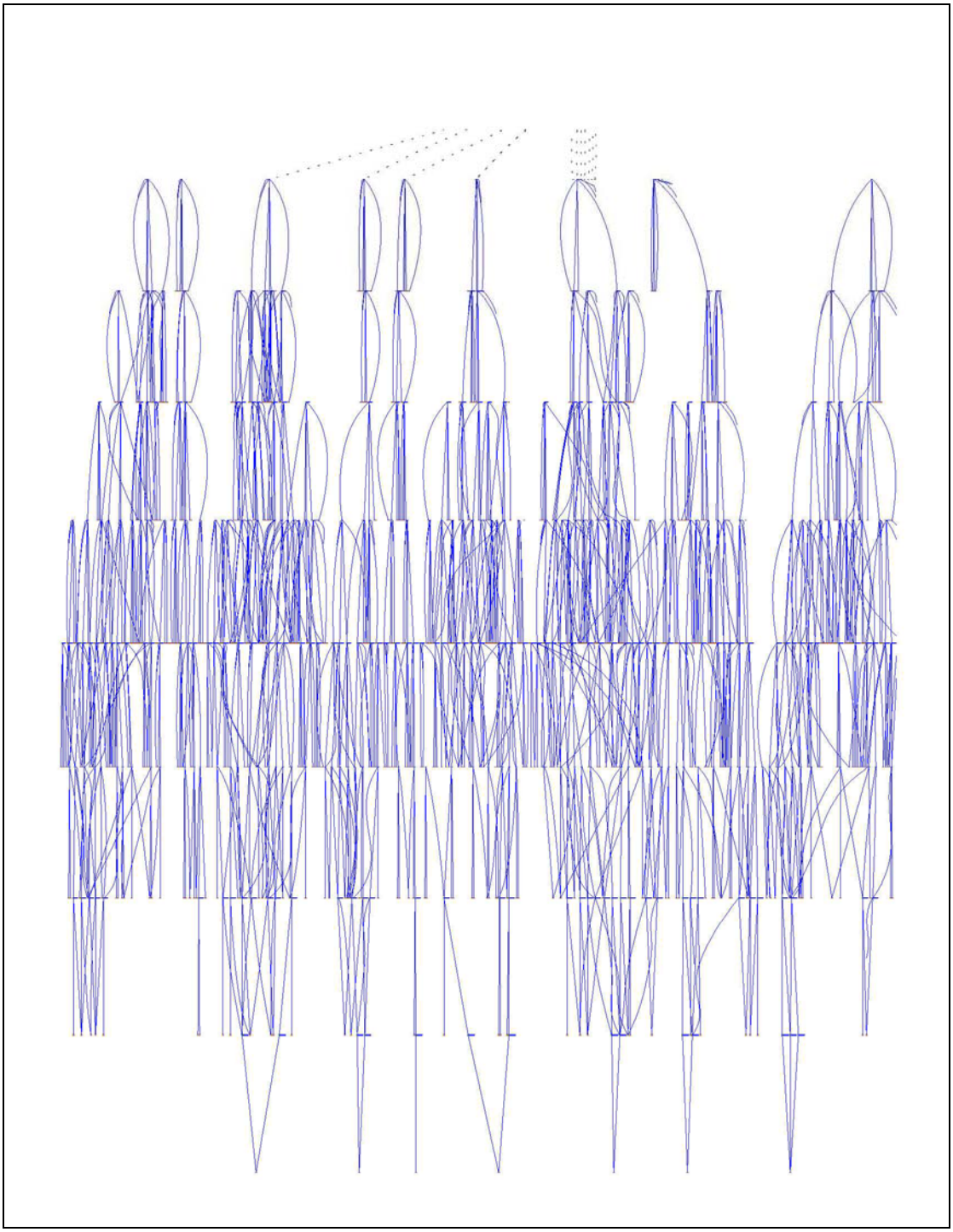

Figure 7