Formal and Intellectual Methods for Systems Security and Resilience. Practicum

Formal and Intellectual Methods
for Systems Security and Resilience
Practicum

O. Pomorova, S. Lysenko, D. Medzatyi, E. Brezhniev
Edited by V.S. Kharchenko

Formal analysis and design for security engineering

Formal methods for the analysis of security protocols

Formal and intellectual methods for system security and resilience

Safety and security analysis of ITS

PRACTICUM

FORMAL AND INTELLECTUAL METHODS FOR SYSTEMS SECURITY AND RESILIENCE

2017

Co-funded by the Tempus Programme of the European Union

Ministry of education and science of Ukraine
Khmelnytskyi national university

Oksana Pomorova, Sergii Lysenko, Dmytro Medzatyi

Formal and Intellectual Methods for System Security and Resilience

Practicum

V. Kharchenko eds.

Project
543968-TEMPUS-1-2013-1-EE-TEMPUS-JPCR
"Modernization of Postgraduate Studies on Security and Resilience for Human and Industry Related Domains"

2017

UDC 004: 504(045)
П22

Authors:
Oksana Pomorova,
Sergii Lysenko,
Dmytro Medzatyi

Oksana Pomorova, Sergii Lysenko, Dmytro MedzatyiПашинцев . **Formal and Intellectual Methods for System Security and Resilience. Practicum** / V. Kharchenko **(edit.).** – Khmelnytskyi: Khmelnytskyi national university, - 2015. – 213 p.

Practical part materials of training course «Formal and Intellectual Methods for System Security and Resilience» which was prepared for TEMPUS «Modernization of Postgraduate Studies on Security and Resilience for Human and Industry Related Domains» (543968-TEMPUS-1-2013-1-EE-TEMPUS-JPCR) masters are posted.

This training course deals with such important questions as development and usage formal methods for designing secure software systems, involvement formal methods for assuring security of computer networks. It also presents some issues on the usage of intelligent systems for security and deals with the questions of the system resilience development. Implementation of the developed training course will improve the quality education and will make graduates successful graduates in the labor market.

This training course is intended for masters and post-graduate students of «Computer engineerings». Also it can used in the study of formal methods for system security and resilience, and may be useful for lecturers with training on relevant courses.

Ref. – 90 items, figures – 51

GLOSSARY

API – Application Programming Interface
AS – Autonomous system
DDS - distributed data store
DES - Data Encryption Standard
DSA - Digital Signature Algorithm
DY - Dolev-Yao
EAL - Evaluation Assurance Levels
ED - Embedded Device
EKE - Encrypted Key Exchange
FADSE - Formal Analysis and Design for Security Engineering
ID – Identifier
IEEE – Institute of Electrical and Electronics Engineers
IP – Internet Protocol
JFK - Just Fast Keying
GA - General Attacker
KAOS – Knowledge Acquisition for automated Specifications
LAN – Local Area Network
MAS - multi-agent systems
MILS - Multiple Independent Levels of Security
MMS - Military Message System
MMU - Memory management unit
NSPK - Needham-Schroeder Public-Key
SCR - Software Cost Reduction
SSH - Secure Shell
SSL - Secure Sockets Layer
TAME - Timed Automata Modeling Environment
TLS - Top-Level Specification
XML - Extensible Markup Language
WAL - write-ahead logging

INTRODUCTION

This book contains the practical part of the materials (laboratory works ana seminars) of the discipline «Formal and Intellectual Methods for System Security and Resilience», prepared for the masters of the TEMPUS project "Modernization of Postgraduate Studies on Security and Resilience for Human and Industry Related Domains" (543968-TEMPUS-1-2013-1-EE-TEMPUS-JPCR). Laboratory works are devoted to the development of formal and intellectual methods for system security and resilient systems.

The manual contains descriptions of laboratory works, seminars, course curriculum.

The first section is devoted to the module of the course "Formal analysis and design for security engineering" and contains theoretical material and a description of the implementation of the two labs. These laboratory works are focused on studying a formal analysis for security engineering in order to capture, organize, and elaborate on security requirements, and on gaining the knowledge and acquire skills in the verifying security down to the source code level.

The second section is devoted to the module of the course "Formal methods for the analysis of security protocols" and contains theoretical material and a description of the a method for verifying security protocols based on an abstract representation of protocols by Horn clauses, as well as a description of the analysis the security protocols under the General Attacker threat model.

The third section is devoted to the module of the course "Formal and intellectual methods for system security and resilience" and contains theoretical material and a description of the a formal goal-oriented approach to development of resilient multi agent system and formalization of the industrial approach to implementing resilient cloud data storage.

For convenience ehe figures, tables and formulas are numbered within each section.

This practicum can be useful for post-graduate students studying in the areas of computer engineering, computer science, and can also be useful to lecturers, leading classes in the relevant disciplines.

Practicum is prepared by Dr. Oksana Pomorova, the head of the system programming department of the Khmelnytskyi national

university, Sergii Lysenko, PhD, associate professor of the system programming department of the Khmelnytskyi national university, and Dmytro Medzatyi, PhD, associate professor of the system programming department of the Khmelnytskyi national university.

The authors are greatly appreciate all partners of the TEMPUS SEREIN1 project consortium for the fruitful collaboration, exchange of experience.

[1] *Этот проект финансируется при поддержке Европейской комиссии. Эта публикация (сообщение) отражает мнения только авторов, и Комиссия не может нести ответственность за любое использование содержащейся в нем информации.*

This project has been funded with support from the European Commission. This publication (communication) reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

1 FORMAL ANALYSIS AND DESIGN FOR SECURITY ENGINEERING

1.1 Laboratory work №1. Formal Analysis and Design for Security Engineering. The Spy Network Case Study

The aim and the task of the laboratory work

The aim of this laboratory work is to consider a formal analysis and design for security engineering, a semi-formal requirements approach, to capture, organize, and elaborate on security requirements. Also it shoud demonstrate, that such formal analysis provides an advantage over a fully formal approach as its goal-directed nature allows enough flexibility while managing to characterize and preserve key security properties that can then be transformed into a proven B representations for further elaboration and refinement at the design and implementation levels.

Task of the work:

- to outline the security requirements of eploying case study, which should demonstrates how these requirements are modeled with Knowledge Acquisition for automated Specifications (KAOS) and transformed to B for further refinement to derive implementation specifications in B.
- to consider thef the application of formal analysis and design for security engineering to the spy network system.

Preparation for laboratory work

- to clarify the aims and objectives;
- to study theoretical material given in the description.

Theoretical material

Introduction

While there are a number of technical approaches on security patterns [1-3], there are few canonical examples from which to formulate a reasonable comparator. According to Fontaine, the security literature does not provide security requirements benchmarks [4]. Rather it has some small examples, which are associated to security models. Unlike many case studies in the security literature, the spy network case study is of a reasonable size; small enough to be manageable and large enough to

be convincing. The spy network case study has been derived from two real case studies:

- The British National Health Service (NHS). The main goal of the system is to protect medical records from illegitimate access to a centralized database.
- The eBay on-line auction web site. It is an example of a typical e-business application with a range of constrains about distributed user behaviors.

The spy network system represents a sample of the category of communication systems that share a common set of security requirements. This assists in verifying the applicability of Formal Analysis and Design for Security Engineering to communication systems that exhibit high security demands.

Case Study Preliminary Problem Statement

The spy network application is aimed at broadcasting secret revelations into a network of spies around the world. Spies are collaborating in teams that achieve a mission each. Each team has a boss. The big boss supervises all missions, allocates spies to missions, and appoints bosses to teams. Spies collect revelations about the enemy and target them to other members of the team working on the mission. The spy who collects a revelation is its author. A spy can be reallocated on another mission, meaning that he goes to another team. Team members should be provided with an uncorrupted copy of the revelation within a certain timeframe. Only spies who are currently allocated to a mission are allowed to know revelations about that mission. Some spies may be malicious spies. Therefore, we need to be sure of the author of a revelation. Workload balancing should be achieved between different spies in the system.

Each spy subscribes to his local service provider for a mailbox in which all his incoming email arrives. Therefore, each spy has a different mail server. Mailboxes are identified by their address. Spy mailbox addresses are very confidential and should not be written down in an insecure location. For security reasons, spies change their mailbox every month. Some spies are old friends and often write to each other outside of the missions duties. Therefore, they memorize others spies' mailbox addresses without having to write them down. Only the owner of a mailbox should be able to access it.

Each spy has at least a few spy friends. Friendship is assumed to last forever. Friends can be in different teams. The author of a revelation is not necessarily the team boss. Revelations are contained in messages that are sent through the email transfer system (asynchronous messages). The email transfer system cannot attack actively, although it could fail to deliver messages or be subject to passive eavesdropping. A revelation is written by an author and read by one or several recipients. A message is sent by a sender and received by a receiver. A message containing a revelation is not necessarily sent directly to the recipients. It could be sent to an intermediate receiver who will send another message containing the same revelation to the recipient, or to another intermediate spy. Revelations are persistent objects, whereas messages are temporary objects that cease to exist upon reception, after their content has been processed. We could think of messages as envelops and of revelations as their content. When a spy collects a new revelation, he sends a message to the team relay. The relay then sends messages to all other members of the team. This assumes that the team relay does know every team member's identity. The team boss appoints the team relay. There is one single relay in each team at a time and every spy knows that fact.

Elaborating Security Requirements with KAOS

Let us outline the elaboration of security requirements for the spy network system with KAOS. The security goals resulting from the elaboration of security requirements are global in the sense that a particular agent cannot enforce them; instead, they apply to the whole system. We will elaborate generic security requirements that are typical in the security domain. This means that they are applicable by analogy to other security domain case studies.

Goal *Maintain* [Security]
InformalDef The system is secure
InstanceOf SecurityGoal

This high level security goal is refined using the traditional classification [5, 6].

Goal *Maintain* [Integrity]
InformalDef Information is guarded against unauthorised update or tampering
InstanceOf SecurityGoal
Refines Security

Goal *Maintain* [Confidentiality]
InformalDef Information is guarded against unauthorised disclosure
InstanceOf SecurityGoal
Refines Security

Goal *Maintain* [Authentication]
InformalDef Agents are really who they claim to be
InstanceOf SecurityGoal
Refines Security

Goal *Achieve* [Availability]
InformalDef Information is guarded against interruption of service
InstanceOf SecurityGoal
Refines Security

Goal *Maintain* [AccessControl]
InformalDef Access control prevents unauthorised access to protected information
InstanceOf SecurityGoal
Refines Security

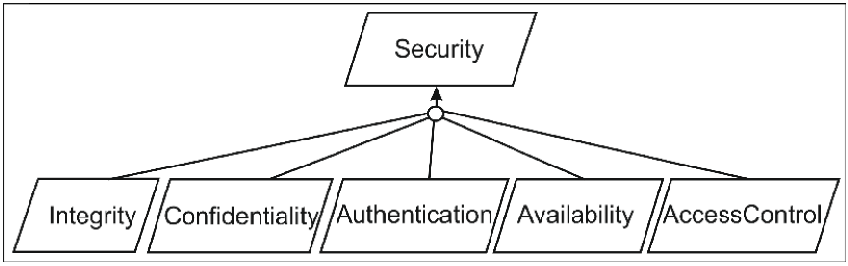

Figure 1.1. Refinement of security goals

This refinement is not complete in that all offspring goals do not necessarily imply the father goal because security properties of a system are of multiple natures. These five subgoals have been chosen because they are known to be the most frequent aspects of a secure system. In order to formally express these security goals, we need to specify a security model for the system, which can be either generic or specific. The literature on security defines generic security models that have to be instantiated to particular systems such as the Bell-LaPadula or Biba models [4]. In order to apply a generic security model to a system, the security requirements of this system need to fit into the model. The Bell-LaPadula or Biba models are appropriate for hierarchical systems like military systems. In these systems, uses at the same level of the hierarchy

have the same rights, which means that privileges are granted to a user class rather than specific users. Many distributed systems do not necessarily fit into this model.

In the context of the spy network case study, a generic security model won't be used, but instead instantiate security goals with domain specific patterns related to the domain will be. All security goals are expressed in terms of the stakeholder's language. For instance, the concept of a Message, which is specific to a particular design is not used. This reflects the fact that these are high level goals and are applicable to any alternative design chosen for the system. In other words, by expressing these security goals, the system is not constrained to fit into a particular security model.

Integrity Goals

The generic pattern for integrity goals can be formalized:

Goal *Maintain* [ObjectCopyAccuracy]
InformalDef Each copy of an object is accurate
InstanceOf IntegrityGoal, AccuracyGoal
FormalDef $\forall$ ob1, ob2 : Object
CopyOf(ob1, ob2) $\Rightarrow$ ob1.Content = ob2.Content

The following heuristic is proposed to instantiate this goal:

For every object copy in the system, the ObjectCopyAccuracy goal should be instantiated. This requires finding out which agent owns the master copy of the object. All other instances of the object will be considered as copies from this master object. In the context of the spy network case study, integrity means that every copy of a revelation should be identical to the original revelation written by the author.

Goal *Maintain* [RevelationIntegrity]
InformalDef A copy of a revelation is identical to the original
InstanceOf IntegrityGoal, AccuracyGoal
FormalDef $\forall$ sp1, sp2 : Spy, re1, re2 : Revelation
Collecting(sp1, re1) $\wedge$ Owning(sp2, re2) $\wedge$ CopyOf(re1, re2)
$\Rightarrow$ re1.Content = re2.Content

The goal RevelationIntegrity is too ideal and cannot be assigned to any agent because the author does not know which revelation content is actually received by the recipient, and conversely the recipient does not know which revelation content the author has sent. Fortunately, this goal is a non-functional one, which means that it is not supposed to be

assigned to a particular agent, and it is global in the system. Maintaining this goal is achieved through the introduction of new functional goals that enforce such non-functional goal.

If every agent assigned to a goal successfully achieves his goal, the integrity of revelations is guaranteed since integrity is implicitly stated in the functional goals. However, stating an explicit goal like InformationIntegrity allows for covering a wider range of agent behaviors in case an agent fails to achieve his goal. We are then able to elaborate strategies that will be refined into strengthened design. This leads to the achievement of system goals even in case of agent failures, which means more robust design. For instance, we could use digital signatures, in which case the recipient is able to verify whether the revelation is intact and is from the purported author. In this operationalization scheme, additional goals are needed to notify the author in case a recipient has received a corrupted copy of a revelation.

Confidentiality Goals

The generic pattern for confidentiality goal can be formalized as follows:

Goal *Maintain* [ObjectConfidentiality]
InformalDef Only agents that verify condition Cond may know the object's attributes
InstanceOf ConfidentialityGoal
FormalDef $\forall$ ag : Agent, ob : Object
$\neg$Cond(ob) $\Rightarrow$ $\neg$Knows(ag, ob.Attribute)

The following heuristic is proposed to instantiate this goal:

For every attribute of an object, express what necessary condition for an agent to know such attribute is. For attributes that do not have such condition, no confidentiality goal is necessary. These conditions depend on domain knowledge. In the context of the spy network case study, confidentiality is refined into two subgoals:

Confidentiality of revelations means that a revelation may be known only by spies working in the mission. Therefore, the generic goal can be instantiated as follows:

Goal *Maintain* [RevelationConfidentiality]
InformalDef The revelation may be known only by spies working in the mission which the revelation is about
InstanceOf ConfidentialityGoal
FormalDef $\forall$ sp : Spy, re : Revelation, te1, te2 : Team
Member(sp, te1) $\wedge$ Targeted(re, te2) $\wedge$ te1 $\neq$ te2 $\wedge \Rightarrow$
$\Rightarrow \neg$ Knows(sp, re.Content)

This goal is non-functional goal that can neither be assigned to the recipient nor the sender. The recipient might want to know the revelation although he should refrain himself from doing so and the sender is not able to control that a wrong target intercepts the message. Therefore, this goal needs to be enforces by further functional goals. Secret keys allows for expressing who is authorized to know a revelation.

Confidentiality of mailboxes means that a spy mailbox address should known only by a friend who can memorize it.

Goal *Maintain* [MailboxConfidentiality]
InformalDef Only friends should know each other's mailbox address
InstanceOf ConfidentialityGoal
FormalDef $\forall$ sp1, sp2 : Spy, ma2 : Mailbox
$\neg$ Friend(sp1, sp2) $\Rightarrow \neg$ Knows(sp1, Subscribed[sp2, ma2])

This goal is too ideal because friendship cannot be controlled by any single agent. It can only controlled by both parties involved. So, we are likely to weaken this goal in different designs.

Authentication Goals

The generic pattern for authentication goals can be formalized as follows:

Goal Maintain[ObjectAuthentication]
InformalDef Agents can verify who has authorship on an object
InstanceOf AuthenticationGoal
FormalDef $\forall$ ag1, ag2 : Agent, ob : Object
AuthorOf(ag1, ob) $\Rightarrow$ Knows(ag2, AuthorOf[ag1, ob])

The following heuristic is proposed to instantiate this goal: for every object, agents should be able to verify its author.

In the context of the spy network case study, authentication means that every revelation is attributable to an author and that the purported author of the revelation is correct. The generic goal can be instantiated as follows:

Goal *Maintain* [RevelationAuthentication]
InformalDef Agents can verify who has authorship on a revelation
InstanceOf AuthenticationGoal
FormalDef $\forall$ sp1, sp2 : Spy, re: Revelation
AuthorOf(sp1, re) $\wedge$ Owning(sp2, re) $\Rightarrow$ Knows(sp2, AuthorOf[sp1, re])

The relationship AuthorOf is equivalent to the relationship Collecting. The spy who collects the revelation is the author. The author mentioned on a revelation can be a forged name, so we will need to verify that the purported author is the real author.

Availability Goals

The generic pattern for availability goals can be formalized as follows:

Goal Achieve [ResourceAvailableForRequest]
InformalDef All agents requests get the resource(s) needed to satisfy the request
InstanceOf AvailabilityGoal
FormalDef $\forall$ ag : Agent, re: Request
RequestIssued(ag, re) $\Rightarrow \Diamond_{<TF}$ ($\exists$ res : Resource) Available(res) $\wedge$
$\wedge$ Sufficient(res, re)

The following heuristic is proposed to instantiate this goal:

For every Achieve goal, determine which resource(s) needs to be available for the goal achievement. In the context of the spy network case study, availability means that revelations are known within a certain timeframe by all other team members. The generic goal can be instantiated as follows:

Goal *Achieve* [RevelationAvailability]
InformalDef All team members know a revelation within 2 hours
InstanceOf AvailabilityGoal
FormalDef <not specified at this stage>

This goal is a quantitative one in which the critical aspect is the timeframe. In this formulation, it is assumed that revelations can be owned within 2 hours. The constant 2 is used as a parameter representing the expected mean time for a revelation transmission

Access Control Goals

The generic pattern for access control goals can be formalized as follows:

Goal *Maintain* [AccessControl]
InformalDef An agent is allowed to access an object if a condition Cond holds
InstanceOf AccessControlGoal
FormalDef ∀ Ag : agent, ob : Object
Accessed(ag, ob) ⇒ Cond(ag, ob)

The following heuristic is proposed to instantiate this goal:

For every object, express what the necessary condition for an agent to access it is. For objects that do not have such condition, no access control goal is necessary. These conditions depend on domain knowledge. In the context of the spy network case study, access control means that a mailbox should be accessed only by its subscribed spy

Goal *Maintain* [MailboxAccessControl]
InformalDef Mailboxes are accessed only by their subscribed spy
InstanceOf AccessControlGoal
FormalDef ∀ sp : Spy, ma : Mailbox
Accessed(ma, sp) ⇒ Subscribed(ma, sp)

Analysis and Resolution of Obstacles and Conflicts for Security Goals

In this subsection, the security goals elaborated in the previous section are refined and analyzed by finding conflicts and obstacles to security, that is, potential attacks. This subsection focuses on finding obstacles to security goals only taking obstacles to the Resolution strategies to potential security attacks are proposed.

Generating Obstacle to the Goal RevelationIntegrity

The goal RevelationForwardedFromRelay is assigned to the relay. In case the relay agent fails to achieve

this goal and modifies the revelation content (maliciously or not), the goal RevelationIntegrity becomes violated as well.

Negating the goal RevelationIntegrity gives the following:

Obstacle RecipientHasCorruptedRevelation
InformalDef Recipient owns a corrupted copy of the revelation
FormalDef ◊ ∃ sp1, sp2 : Spy, re1, re2 : Revelation
Collecting(sp1, re1) ∧ Owning(sp2, re2) ∧ CopyOf(re1, re2) ∧
∧ re1.Content ≠ re2.Content

The following object model increment is required:

Relationship Corrupted
InformalDef A revelation copy is not intact with respect to the original revelation
Links Revelation {**Role** Caracterised, **Card** 0:N}
DomInvar
∀ re1, re2 : Revelation
Corrupted(re1) ∨ Corrupted(re2) ⇒ CopyOf(re1, re2) ∧ re1.Content ≠ re2.Content
∀ sp1, sp2, sp3 : Spy, re1, re2 : Revelation, me1, me2 : Message
Receiving(sp2, me, sp1) ∧ Sending(sp2, me, sp3) ∧ About(me1, re1)
∧ About(me2, re2) ∧ CopyOf(re1, re2) ∧ re1.Content ≠ re2.Content ⇒
⇒ Corrupted(re2)
∀ sp : Spy, re : Revelation
Collecting(sp, re) ⇒ ¬Corrupted(re)

Regressing through the domain properties, we get:

Obstacle RecipientHasCorruptedRevelation
InformalDef An agent has sent a different revelation than the one he has received
FormalDef ◊ ∃ sp1, sp2 : Spy, re1, re2 : Revelation
Collecting(sp1, re1) ∧ Owning(sp2, re2) ∧
[Receiving(sp2, me, sp1) ∧ Sending(sp2, me, sp3) ∧ About(me1, re1)
∧ About(me2, re2) ∧ CopyOf(re1, re2) ∧ re1.Content ≠ re2.Content] ∨
∨ Corrupted(re1)

Because of the third domain property, we know that the revelation has been corrupted by sp2 (the relay) rather than by sp1 (the author). If the relay changes the content of the revelation, the RevelationIntegrity goal is violated. A strong mitigation would be achieved through the following goal:

Goal *Achieve* [WholeTeamInformedWhenCorrupted]
InformalDef If a spy collects a revelation, eventually all team members will own it
even if a previous copy of the revelation was corrupted
FormalDef ∀ sp1, sp2 : Spy, re1, re2 : Revelation, te : Team
Collecting(sp1, re1) ∧ Member(sp1, te) ∧ Member(sp2, te)
∧ Owning(sp2, re2) ∧ CopyOf(re2, re1) ∧ Corrupted(re2)
⇒ ◊ (∃ re3 : Revelation) Owning(sp2, re3) ∧ CopyOf(re3, re1)
∧ ¬ Corrupted(re3)

This goal refined as follows:

Goal *Maintain* [CorruptedRevelationKnownByAuthor]
InformalDef The author knows when another spy owns a corrupted copy of his revelation
Refines WholeTeamInformedWhenCorrupted
FormalDef ∀ sp1, sp2 : Spy, re1, re2 : Revelation
Collecting(sp1, re1) ∧ Owning(sp2, re2) ∧ CopyOf(re2, re1)
∧ Corrupted(re2) ⇒ Knows(sp1, Corrupted[re2])

Goal *Achieve* [AuthorResendWhenKnowsCorrupted]
InformalDef If a spy believes that a revelation was corrupted, another revelation copy that is not corrupted will be owned by the team members
Refines WholeTeamInformedWhenCorrupted
FormalDef ∀ sp1, sp2 : Spy, te : Team, re1, re2 : Revelation
Collecting(sp1, re1) ∧ Member(sp1, te) ∧ Member(sp2, te)
∧ Owning(sp2, re2) ∧ CopyOf(re2, re1) ∧ Belief(sp1,Corrupted[re2])
⇒ ◊ (∃ re3 : Revelation) Owning(sp2, re3) ∧ CopyOf(re3, re1)
∧ ¬ Corrupted(re3)

The goal AuthorResendWhenKnowsCorrupted needs to be refined the same way as the goal WholeTeamInformed. In this case, the revelation author could send the revelation again hoping that it was only a temporary error and that the revelation will not get corrupted this time. The goal CorruptedRevelationKnownByAuthor needs to be refined as follows:

Goal *Maintain* [CorruptedRevelationKnownByReceiver]
InformalDef A receiver knows when a revelation is corrupted
Refines CorruptedRevelationKnownByAuthor
FormalDef ∀ sp1, sp2 : Spy, re1, re2 : Revelation
Collecting(sp1, re1) ∧ Owning(sp2, re2) ∧ CopyOf(re2, re1)
∧ Corrupted(re2) ⇒ Knows(sp2, Corrupted[re2])

Goal *Maintain* [AuthorKnowsCorruptedWhenReceiverKnows]
InformalDef If a receiver of a revelation knows
that it is corrupted, its author will know it
Refines CorruptedRevelationKnownByAuthor
FormalDef ∀ sp1, sp2 : Spy, re1, re2 : Revelation
Collecting(sp1, re1) ∧ Owning(sp2, re2) ∧ CopyOf(re2, re1)
∧ Belief(sp2, Corrupted[re2])
⇒ ◊ Belief(sp1, Corrupted[re2])

The goal AuthorKnowsCorruptedWhenReceiverKnows will be refined using notifications in a similar way as the goal WholeTeamInformed as shown below.

Resolving Obstacles to the Goal RevelationIntegrity

Van Lamsweerde defined some patterns to resolve obstacles early in requirements and accommodate these solutions in the requirements model [7]. In the security context, obstacles represent security threats to the system that need to be resolved using security patterns [8]. For example, data integrity is normally preserved using digital signatures, so a way to resolve the obstacles to the RevelationIntegrity goal is to employ digital signatures in the goal operationalization of the RevelationIntegrity goal. The early accommodation of this solution during requirements analysis allows for reflecting the impact of this solution on the rest of the requirements that have dependency on this RevelationIntegrity goal.

The RevelationIntegrity goal is operationalized such that each revelation has a signature that depends on the revelation text and the author's identity. With a public key scheme, there are different keys for signing (private key) and verifying (public key). Verification allows the receiver to check whether the message is intact. Each spy can have a list of public keys of every other spy including those not in his team.

When he receives a revelation from a spy, he can verify that the revelation is from the purported author if the key is accurately associated

with the author. If a spy has several public/private keys, they need to be identified. For simplicity, let's assume in the coming illustration that each spy has only one key pair. The signature of the message could be used to deduce that the revelation is not corrupted with respect to the public key used to verify it as defined in the goal RevelationVerifiedWhenReceived. In case the verification indicates that the revelation is corrupted, it could also be the case that it is actually the signature that is corrupted and the revelation is corrupted. The goal CorruptedRevelationKnownByReceiver can be refined into the goals RevelationSignedWhenSent and RevelationVerifiedWhenReceived that mandate the signing of the revelation at the sender side and the verification of that signature at the receiver end as follows:

Goal *Maintain* [RevelationSignedWhenSent]
InformalDef Revelations are signed with the author's private key
Refines CorruptedRevelationKnownByReceiver
FormalDef ∀ sp1, sp2 : Spy, re1, re2 : Revelation
Collecting(sp1, re1) ∧ Owning(sp2, re2) ∧ CopyOf(re2, re1)
⇒ (∃ prk : PrivateKey, puk : PublicKey, si : Signature)
Signed(re1, si, prk) ∧ Owning(sp2, si)
∧ KeyPair(puk, prk) ∧ Owning(sp2, puk)

Goal *Maintain* [RevelationVerifiedWhenReceived]
InformalDef The receiver knows whether the revelation is genuine with signature verification
Refines CorruptedRevelationKnownByReceiver
FormalDef ∀ sp1, sp2 : Spy, me1, me2 : Message, re1, re2 : Revelation, prk : PrivateKey, puk : PublicKey
Collecting(sp1, re1) ∧ Owning(sp2, re2) ∧ CopyOf(re2, re1)
∧ Signed(re1, si, prk) ∧ Owning(sp2, si)
∧ ¬ Verified(re2, si, puk).Genuine
∧ KeyPair(puk, prk) ∧ Owning(sp2, puk)
⇒ Knows(sp2, Corrupted[re2])

The goals RevelationSignedWhenSent and RevelationVerifiedWhenReceived are assigned to the sender and the receiver agents respectively. The operations SignRevelation and VerifyRevelation operationalize the two goals as indicated below. The goal graph showing the refinement of the resolution of the RevelationIntegrity obstacles is illustrated in Figure 1.2.

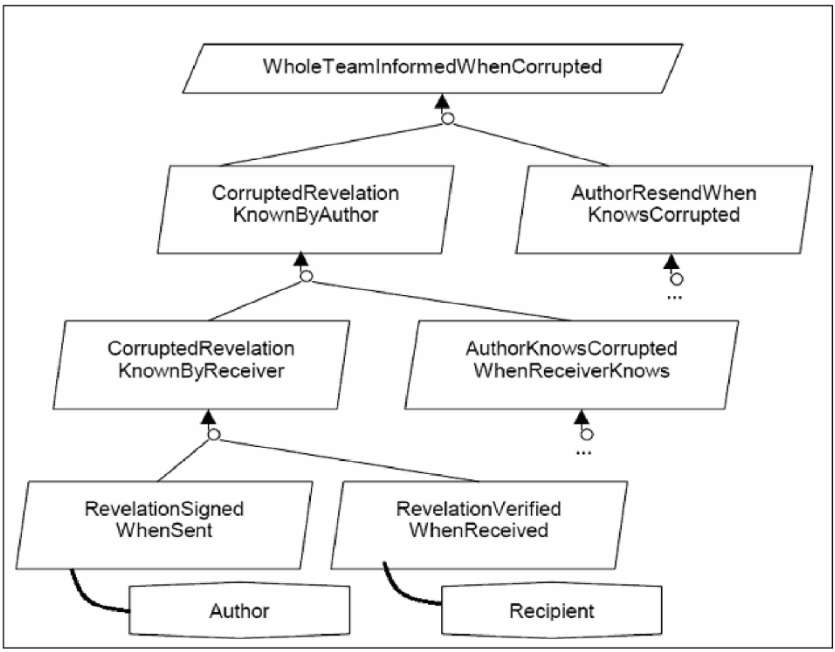

Figure 1.2. Refinement of the Integrity obstacle resolution

The above two goals introduced to resolve the obstacles of the RevelationIntegrity goal yield some increments in the object model in order to accommodate the digital signature solution. Assuming that public key infrastructure is employed to carry out digital signatures, the entities of a PrivateKey, PublicKey and KeyPair are needed as well as entities modeling the signature itself and the relationships Signed and Verified. The object model increments are defined as follows:

Entity Signature
InformalDef A signature depends on a text and a private key and is used to validate it

Entity Key
InformalDef A key is a secret magic word that allows to understand some secret information

Entity PrivateKey
InformalDef A private key is the key in an asymmetric key scheme that is kept secret by its author
IsA Key

Entity PublicKey
InformalDef A public key is the key in an asymmetric key scheme that is broadcasted by its author
IsA Key

Relationship KeyPair
InformalDef A key pair is composed of a public key and a private key
Links PublicKey {**Role** Public, **Card** 1:1}
PrivateKey {**Role** Private, **Card** 1:1}
DomInvar $\forall$ prk : PrivateKey, puk : PublicKey
KeyPair(puk, prk) $\Rightarrow$
($\exists$ sp : Spy) CreatorOf(sp, puk) $\wedge$ CreatorOf(sp, prk)

Relationship Signed
InformalDef A revelation is signed with a private key, resulting in a signature
Links Revelation {**Role** Signed, **Card** 1:N}
Signature {**Role** Signature, **Card** 1:N}
PrivateKey {**Role** Key, **Card** 1:N}

Relationship Verified
InformalDef A revelation is verified with a public key and a given signature
Links Revelation {**Role** Verified, **Card** 1:N}
Signature {**Role** Verifies, **Card** 1:N}
PublicKey {**Role** Key, **Card** 1:N}
Has Genuine : Boolean
DomInvar
$\forall$ sp : Spy, re : Revelation, puk : PublicKey, prk : PrivateKey, si : Signature
Signed(re, si, prk) $\wedge$ Owning(sp, si) $\wedge \neg$ Verified(re, si, puk).Genuine
$\wedge$ KeyPair(puk, prk) $\wedge$ Owning(sp, puk)
$\Rightarrow$ Knows(sp, $\neg$ Signed[re, si, prk]) $\wedge$ Knows(sp, Corrupted[re])
$\forall$ sp : Spy, re : Revelation, puk : PublicKey, prk : PrivateKey, si : Signature
Signed(re, si, prk) $\wedge$ Owning(sp, si) $\wedge$ Verified(re, si, puk).Genuine
$\wedge$ KeyPair(puk, prk) $\wedge$ Owning(sp, puk)
$\Rightarrow$ Knows(sp, Signed[re, si, prk]) $\wedge$ Knows(sp, $\neg$ Corrupted[re])

Relationship CreatorOf
InformalDef A spy is the creator of a key
Links Spy {**Role** Creates, **Card** 1:1}
Key {**Role** Created, **Card** 1:1}

In the goal CorruptedRevelationKnownByReceiver, it is stated that the author's public key is owned by the receivers of the revelation. It means that keys have to be distributed like revelations with the difference that public keys last for the entire life of a spy. The key distribution will also yield a goal for their broadcasting similar to the goal WholeTeamInformed:

Goal *Maintain* [BroadcastPublicKey]
InformalDef Spies in the same teams should own each other's key
FormalDef $\forall$ sp1, sp2 : Spy, puk : PublicKey, te : Team
CreatorOf(sp1, puk) $\wedge$ Member(sp1, te) $\wedge$ Member(sp2, te)
$\Rightarrow$ Owning(sp2, puk)

A new type of message is introduced, PublicKeyNotif to be sent by a spy to inform other people of his public key.

Entity PublicKeyNotif
InformalDef Announcement from a spy giving out his public key
IsA Message
Has Spy : Spy
PubKey : PublicKey

Because a private key identifies a spy, it needs to be owned only by its creator:

Goal *Maintain* [SafePrivateKey]
InformalDef A spy owns a private key only if he is the owner of this key
FormalDef $\forall$ sp : Spy, prk : PrivateKey
Owning(sp, prk) $\Rightarrow$ CreatorOf(sp, prk)

The above object model increment has propagated the impact of introducing digital signature as a solution to the obstacles of the RevelationIntegrity goal. The rest of the security goals that relate to RevelationIntegrity feel the impact of the digital signature solution through the update of the object model that mediates the interaction among goals. A graphical summary of all security goals after applying obstacle analysis is illustrated in Figure 1.3.

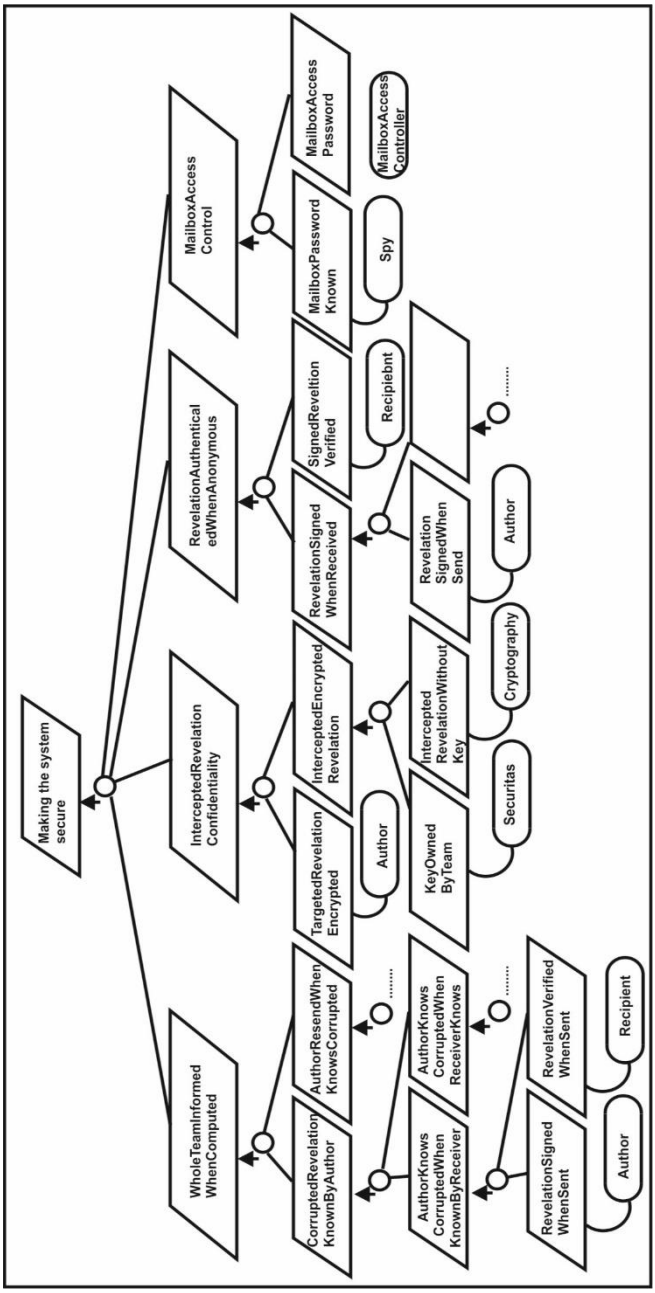

Figure 1.3. Security goal graph for the spy network system

Figure 1.4 shows the part of the object model involved in security goals.

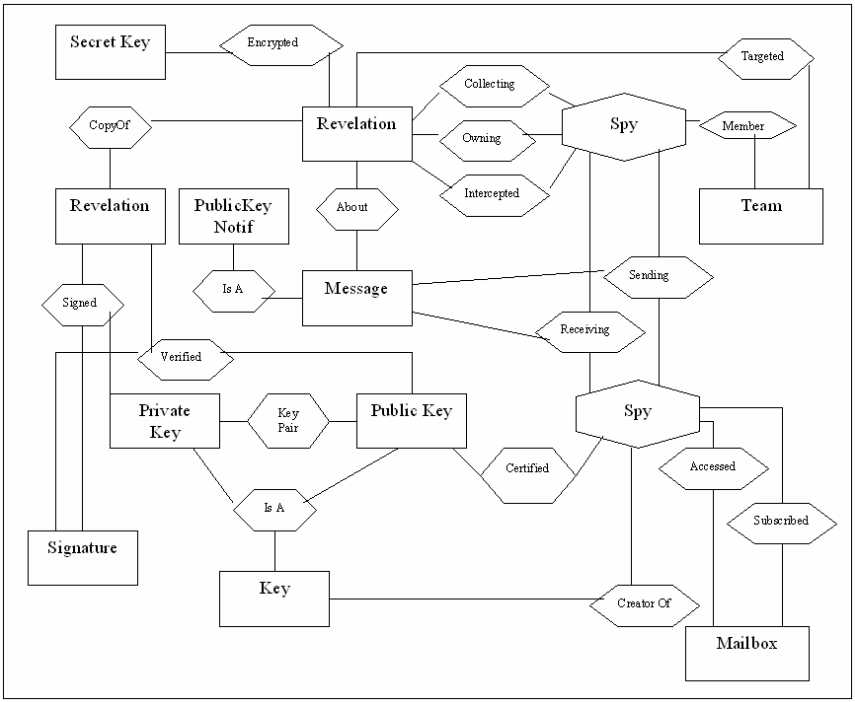

Figure 1.4. Partial object model involved in security goals

Requirements at the very bottom of the goal graph need to be operationalized in order to complete the KAOS model. KAOS operations are means for agents in the software-to-be to achieve their assigned requirements. The goal graph of the security requirements for the spy network system in Figure 1.3 is big and complicated, so we have summarized the security operations along with the fundamental security goals in Figure 1.5.

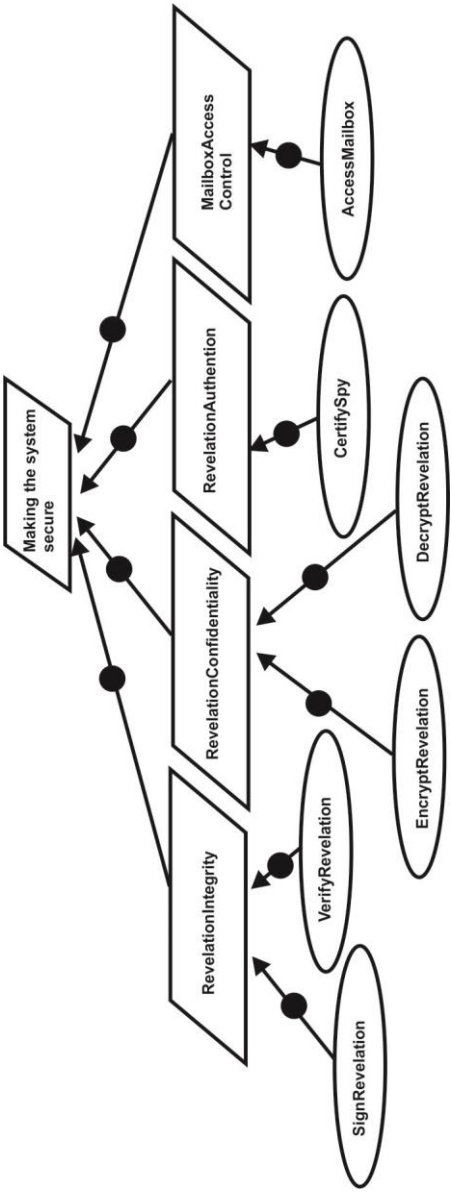

Figure 1.5. Summary of the KAOS Operations for the Spy Network Security Goal Graph

Figure 1.5 shows six security operations. The SignRevelation and VerifyRevelation operations realize the revelation integrity goals. The EncryptRevelation and DecryptRevelation goals realize the revelation confidentiality goals. The CertifySpy operation realizes the revelation authentication goals and the AccessMailbox realizes the mailbox access control goals. The following formal definition of each operation shows the operation name (highlighted), the input parameters (Input), the output parameters (Output), the precondition (DomPre) that must be true prior to the operation execution, the post condition (DomPost) that must be true after the operation finishes execution, and the goals that this operation are prerequisite to their achievement (ReqPreFor). The SignRevelation operation is responsible for achieving the goal RevelationIntegrity and its subgoals. This operation is called when a spy sends a revelation to another spy in order to protect the integrity of the revelation against malicious acts. The operation takes the spy signing the revelation, the revelation to be signed and the private key with which the revelation is signed as input parameters and returns the digital signature as an output.

Operation SignRevelation
Input Spy{arg sp}, Revelation{arg rev}, Privatekey {arg pk}
Output Signature {res sig}
DomPre $\neg$ ($\exists$ rev1:Revelation) Signed(rev1, sig, pk)
DomPost ($\exists$ rev2:Revelation) Signed(rev2, sig, pk)
ReqPreFor RevelationSignedWhenSent
CreatorOf(sp, pk)

The VerifyRevelation operation is responsible for achieving the goal RevelationIntegrity and its subgoals. This operation is called when a spy receives a revelation from another spy in order to verify that the received revelation is not tampered with while in transit. The operation takes the spy verifying the revelation, the revelation to be verified and the public key with which the revelation is verified as input parameters and returns a boolean as an output to indicate whether the revelation is verified correct or not.

Operation VerifyRevelation

Input Spy{arg sp}, Revelation{arg rev}, Publickey {arg pk}, Signature {arg sig}
Output Boolean {res verified}
DomPre ¬(∃ rev1:Revelation) Verified(rev1, sig,Pk)
DomPost (∃ rev2:Revelation) Verified(rev2, sig, pk)
ReqPreFor RevelationVerifiedWhenReceived
Knows(sp, pk)
ReqPreFor RevelationDecryptedWhenReceived
(∃ msg:Message, prk: PrivateKey)
Decrypted(msg, prk) ^ CreatorOf(sp, prk)

The EncryptRevelation operation is responsible for achieving the goal RevelationConfidentiality and its subgoals. This operation is called when a spy sends a revelation to another spy in order to protect the confidentiality of the revelation against malicious acts. The operation takes the spy encrypting the revelation, the revelation to be encrypted and the public key with which the revelation is encrypted as input parameters and returns the encrypted revelation as an output.

Operation EncryptRevelation
Input Spy{arg sp}, Revelation{arg rev}, Publickey {arg pk}
Output Message {res msg}
DomPre ¬(∃ rev1:Revelation) Encrypted(rev1, pk)
DomPost (∃ rev2:Revelation) Encrypted(rev2, pk)
ReqPreFor RevelationEncryptedWhenSent
Knows(sp, pk)
ReqPreFor RevelationSignedWhenSent
(∃ prk: PrivateKey, sig:Signature)
Signed(rev, sig, prk) ^ CreatorOf(sp, prk)
ReqPostFor RevelationEncyptedWhenSent
About(msg, rev)

The DecryptRevelation operation is responsible for achieving the goal RevelationConfidentiality and its subgoals. This operation is called when a spy receives a revelation to another spy in order to get the content of the encrypted revelation. The operation takes the spy receiving the revelation, the encrypted revelation to be decrypted and the private key

with which the revelation is decrypted as input parameters and returns the revelation content as an output.

Operation DecryptRevelation
Input Spy{arg sp}, Message{arg msg}, Privatekey {arg pk}
Output Revelation {res rev}
DomPre ¬(∃ msg1:Message) Decrypted(msg1, pk)
DomPost (∃ msg2:Message) Decrypted(msg2, pk)
ReqPreFor RevelationDecryptedWhenReceived
CreaterOf(sp, pk)
ReqPostFor RevelationDecryptedWhenReceived
Knows(sp, rev.Content)

The CertifySpy operation is responsible for achieving the goal RevelationAuthentication and its subgoals. This operation is called when a spy receives a revelation from another spy in order to authenticate the sender of the revelation. The operation takes the spy sending the revelation, and the public key of the sender and returns a boolean as an output indicating whether the revelation comes from a certified spy or not.

Operation CertifySpy
Input Spy{arg sp}, Publickey{arg pk}
Output Boolean {res Authenticated}
DomPre ¬Certified(sp, pk)
DomPost Certified(sp, pk)
ReqPreFor RevelationVerifiedWhenReceived
(∃ rev:Revelation,, sig:Signature)
Verified(rev, sig, pk) ^ CreatorOf(sp, pk)

The AccessMailbox operation is responsible for achieving the goal MailboxAccessControl and its subgoals. This operation is called when a spy tries to access the mailbox in which he/she is subscribed. The operation takes the spy trying to access his/her mailbox, the mailbox being accessed, and the password of the mailbox and returns a boolean as an output indicating whether access to the mailbox is allowed or denied based on the provided password.

Operation AccessMailbox
Input Spy{arg sp}, Mailbox{arg ma}, Password {arg pa}
Output Boolean {res accessed}
DomPre ¬ Accessed(ma, sp)
DomPost Accessed(ma, sp)
ReqPreFor MailboxAccessControl
Subscribed(ma, sp)
ReqPreFor MailboxAccessedWithPassword
pa = ma.Password

The above operations are transformed to B in the coming subsection to construct the initial B machine that is further refined inside B using the B refinement mechanism to derive design specifications and generate implementation. The rational for transforming KAOS operations to B while not transforming the rest of the goal graph is that operations sums up all the behaviors that agents need to have to fulfill their requirements, which are the leaf goals in the goal graph. The mechanism for constructing the goal graph shows that high level goals are refined using AND/OR refinement steps until leaf goals are derived meaning that the fulfillment of leaf goals implies the fulfillment of the higher level goals in the goal graph. Therefore, it is safe to only transform KAOS operations used to express behaviors of agents that perform them to fulfill the leaf goals in the goal graph without compromising the completeness and consistency properties of the requirements model.

Transforming the Spy Network Security Goal Graph to B

Modeling the security requirements of the spy network system with KAOS has defined the goals, the agents responsible for achieving these goals and the means to achieve these goals in the form of the KAOS operations. In order to go further with the security-specific elements from the requirements phase to the design phase, the KAOS requirements model need to be transformed to a design elaboration language, which is the B language in FADSE [1], The transformation focuses on both the KAOS security operations and the entities of the partial object model involved in security goals. The transformation scheme provides a means to bridge the gap between security requirements and their realization in formal design. The value of the transformation scheme is in stepping

further from the relaxed formality of the KAOS requirements model in which requirements are wellorganized and reasoned about to more rigid formality in the initial B model that is further refined for design. This means that without the transformation scheme, the variance of formality between requirements and design would not have been possible. This is evidenced in the formal security engineering literature in which rigid formality applies to all the phases of development starting for specifying requirements to deriving implementation like employing the Z or the VDM formal languages for manipulating security concerns. Applying formal languages to requirements specifications has the disadvantages of increasing the cost and complexity of development, using formal languages that are very specific to model requirements that usually have lots of unknowns that cannot be specified formally, and lacking built-in constructs for threat analysis and mitigation. The Goal Graph Analyzer tool that automates the transformation from KAOS to B parses the XML model produced by the Objectiver tool and representing the KAOS security requirements model for the spy in order to construct the initial B abstract model equivalent to the KAOS model. The initial B model is manipulated using the B-Toolkit, which is one of the two most famous commercial tools for B development as a tool to develop B model and refine it to derive design specifications and implementation. The abstract B machine for the spy network system is illustrated in the below code. The spy network system is represented as an abstract machine called SpyNetwork parameterized with the maximum number of spies allowed in the system. The machine represents the spies as a set in the Variables section that model the system state as highlighted in the below code. The Spies' attributes are also modeled as part of the Variables section with the Invariant stating the types of each attribute. The machine invariant states the constraints on the machine state variables. The SpyNetwork machine represents each of the six KAOS operations illustrated in Figure 1.5 as a B operation as highlighted below. KAOS operations preconditions are directly mapped to preconditions of their corresponding B operations since both are defined in first-order predicate logic. The KAOS definition of operations provides the interface of the operation with which its clients (callers) will call it and that is the task of the requirement analysis phase. The abstract definition of each operation behavior is the responsibility of the early design phase. This means that the generated B abstract machine from the Goal Graph Analyzer needs to

be augmented with the abstract specifications for each operation as shown below. The abstraction specification of each operation is further refined using the B refinement mechanism to make the definition more concrete through adding more details and removing the non-determinism in the abstract definition.

MACHINE SpyNetwork (maxSpies)
CONSTRAINTS maxSpies : 1..10000
SEES StrTokenType, Bool_Type
DEFINITIONS SPY== 1.. maxSpies

VARIABLES
spies, spyId,spyName, spyMailboxPassword, spyPublicKey, spyPrivateKey,
key, signature –-used as a placeholder for use in local variables

INVARIANT
spies <:SPY &
spyId : spies >-> NATURAL1 &
spyName : spies --> STRTOKEN &
spyMailboxPassword : spies >-> STRTOKEN &
spyPublicKey : spies >-> STRTOKEN &
spyPrivateKey : spies >-> (STRTOKEN - spyPublicKey) &
spyId >< spyName>< spyMailboxPassword ><spyPublicKey><spyPrivateKey:
spies >-> (NATURAL1 >< STRTOKEN >< STRTOKEN >< STRTOKEN >< STRTOKEN) &
key : STRTOKEN & signature : STRTOKEN

OPERATIONS

NewSpy(identity, name, mailboxPassword, publicKey, privateKey) =
  PRE

    identity: NATURAL1 & name: STRTOKEN & mailboxPassword: STRTOKEN &
    publicKey: STRTOKEN & privateKey: STRTOKEN &
    (identity, name, mailboxPassword, publicKey, privateKey)/:
    ran(spyId >< spyName >< spyMailboxPassword >< spyPublicKey >< spyPrivateKey) &

```
        spies /= SPY

    THEN

        ANY newSpy WHERE newSpy : SPY – spies THEN
      spies := spies \/ {newSpy} || spyId(newSpy) := identity ||
      spyName(newSpy) := name ||
        spyMailboxPassword(newSpy) := mailboxPassword ||
        spyPublicKey(newSpy) := publicKey ||
        spyPrivateKey(newSpy) := privateKey
    END
END;


  found, spyDetails <-- getSpy(identity) =
    PRE identity : NATURAL1 THEN
    IF (identity) : ran(spyId) THEN
      spyDetails := (spyId >< spyName >< spyMailboxPassword ><
                        spyPublicKey >< spyPrivateKey)~ (identity) ||
        found := TRUE
    ELSE
        spyDetails : SPY ||
        found := FALSE
    END
END;

full <-- spyNetworkFull =
BEGIN
    full := bool(spies = SPY)
END;

encyptedRevelation <-- encryptRevelation(revelation, identity) =
    PRE revelation : STRTOKEN & identity : NATURAL1 THEN
    IF (identity) : ran(spyId) THEN
        key := spyPublicKey(identity) ||
        signature := signRevelation(revelation, identity);
        encyptedRevelation : (signature; key) >-> STRTOKEN
    ELSE
        encyptedRevelation :: STRTOKEN
    END
```

```
END;

revelation <-- decryptRevelation(ecryptedRevelation, identity) =
    PRE ecryptedRevelation : STRTOKEN & identity : NATURAL1 THEN
    IF (identity) : ran(spyId) THEN
        key := spyPrivateKey(identity) ||
        revelation : ecryptedRevelation >-> STRTOKEN
    ELSE
        revelation :: STRTOKEN
    END
END;

signature <-- signRevelation(revelation, identity) =
  PRE revelation : STRTOKEN & identity : NATURAL1 THEN
    IF (identity : ran(spyId)) THEN
        key := spyPrivateKey(identity) ||
        signature : revelation >-> STRTOKEN
    ELSE
        signature :: STRTOKEN
    END
END;

verified <-- verifyRevelation(revelation, identity, signature) =

  PRE
      revelation : STRTOKEN & identity : NATURAL1 & signature : STRTOKEN
  THEN
  IF (identity) : ran(spyId) THEN
      key := spyPublicKey(identity) ||
      IF (signature == revelation(key)) THEN -- how to indicate the application of key to
  revelation
          verified := TRUE
      ELSE
          verified := FALSE
      END
  ELSE
    verified := FALSE
  END
END;
```

```
authenticated <-- certifySpy(identity, publicKey) =
  PRE identity : NATURAL1 & publicKey : STRTOKEN THEN
  authenticated := bool(spyPublicKey(identity) = publicKey)
END;

accessAllowed <-- accessMailbox(identity, password) =
  PRE identity : NATURAL1 & password : STRTOKEN THEN
    IF (identity : ran(spyId)) THEN
          IF (password == spyMailboxPassword(identity)) THEN
            accessAllowed := TRUE
          ELSE
            accessAllowed := FALSE
       END
    ELSE
       accessAllowed := FALSE
    END
END;
```

To illustrate the idea of the abstract definition of operation behavior that augments the generated B machine from the Goal Graph Analyzer Tool, let's describe the definition of the encryptRevelation operation as an example. The definition first checks on whether the spy whose identity is used to encrypt the message belongs to the set of spies in the network. If the spy identity is verified, the public key of the spy is retrieved from the set of public keys stored in the variables section using the spy identity. The revelation is signed before encrypted by calling the signRevelation operation and finally the encrypted revelation is abstractly defined as another string value calculated from the original revelation. The abstract definition of the encrypted revelation allows for multiple concrete definitions of how to encrypt the original revelation depending on the design decision made in the coming refinement steps on which encryption algorithm is used.

Derivation of Design and Implementation

The initial B machine is refined using the B refinement mechanism to enforce design decisions. The first refinement step is classified as data refinement concerned with refining the representation of the machine variables that reflect the system state. The first refinement step represents the pool of spies as an array of spies and it will be shown how the

security operations are refined accordingly. From the traceability perspective, the data refinement step does not directly address the realization of a specific security requirement in the system. It rather concentrates on building a concrete data structure representing the internal system state in a form realizable by programming languages while implementation is generated. The first refinement is by convention named with the same name of the machine it refines with an R appended to the machine name. The invariant of the refining machine should be linked to the variables of the refined machine by means of linking invariant that describes the relationship between the state spaces of the two machines [10]. The linking invariant is used to generate proof obligations that specify which proofs need to be discharged in order to prove that the refining machine does not violate any of the constraints of the refined machine; therefore, proves correctness of development and preservation of security properties. The linking invariant in the first refinement step of the spy network system is dom(spiesr) = spies meaning that the set spies is precisely the domain of the function spiesr since this gives the index of the set of elements that appear in the array. The elements in the spiesr array are the Ids of all the spies in the system. The rest of the spy's attributes are linked to each spy through his/her Id. The operations are defined on the variable spiesr. Operations are required to work only within their preconditions given in the abstract machine, so those preconditions are assumed to hold for the refined operations. Therefore, the type information of the input variables and the other requirements on them do not need to be repeated in the refinement machine. The highlighted parts of the first refinement machine reflects the implication of the data refinement on the abstract definition of the operations.

REFINEMENT SpyNetworkR
REFINES SpyNetwork
SEES StrTokenType, Bool_TYPE

VARIABLES
spiesr, spyNamer, spyMailboxPasswordr, spyPrivatekeyr, spyPublickeyr

INVARIANT
spiesr : 1.. maxSpies >-> spyId & dom(spiesr) = spies &
spyNamer : spiesr >-> STRTOKEN & ran(spyNamer) = spyName &

```
spyMailboxPasswordr : spiesr >-> STRTOKEN &
ran(spyMailboxPasswordr) = spyMailboxPassword &
spyPrivatekeyr : spiesr >-> (STRTOKEN – spyPublickey) &
ran(spyPrivatekeyr) = spyPrivatekey &
spyPublickeyr : spiesr >-> (STRTOKEN- spyPrivatekey) &
ran(spyPublickeyr) = spyPublickey
INITIALIZATION Spiesr := (1..maxSpies) >< {0} -- all spies ids are initialized to 0

OPERATIONS

encyptedRevelation <-- encryptRevelation(revelation, identity) =
      VAR key IN THEN
      IF (spiesr~(identity): dom(spiesr)) THEN
            key := spyPublickeyr(identity) ||
            signature := signRevelation(revelation, identity);
            encyptedRevelation : (signature; key) >-> STRTOKEN
      ELSE
            encyptedRevelation :: STRTOKEN
      END
END;

revelation <-- decryptRevelation(ecryptedRevelation, identity) =
      VAR key IN
      IF (spiesr~(identity): dom(spiesr)) THEN
            key := spyPrivatekeyr(identity) ||
            revelation : ecryptedRevelation >-> STRTOKEN
      ELSE
            revelation :: STRTOKEN
      END
END;

signature <-- signRevelation(revelation, identity) =
  VAR pk IN
  IF (spiesr~(identity): dom(spiesr)) THEN
    pk := spyPrivatekeyr(identity);
    signature : revelation >-> STRTOKEN
  ELSE
    signature:= EmptyStringToken;
  END
END;
```

```
verified <-- verifyRevelation(revelation, identity, signature) =
  VAR key IN
  IF (spiesr~(identity) : dom(spiesr)) THEN
        key := spyPublickeyr(identity) ||
        IF (signRevelation(revelation, identity) == signature) THEN
            verified := TRUE
        ELSE
            verified := FALSE
        END
  ELSE
    verified := FALSE
  END
END;

authenticated <-- certifySpy(identity, publicKey) =
  VAR key IN
  authenticated := bool(spyPublickeyr(identity) = publicKey)
END;

accessAllowed <-- accessMailbox(identity, password) =
  IF (spiesr~(identity) : dom(spiesr)) THEN
    IF (password == spyMailboxPasswordr(identity)) THEN
          accessAllowed := TRUE
    ELSE
          accessAllowed:= FALSE
    END
  ELSE
     accessAllowed:= FALSE
  END
END;
```

The above B code that represents the first refinement step maintains the abstract definition of the machine operations and reflects the new data representation of machine variables through using the arrays spiesr, spyNamer, spyMailboxPasswordr, spyPrivatekeyr, spyPublickeyr instead of the abstract sets. The second refinement step is a procedural refinement that makes design decisions about the security algorithms used for preserving the revelation integrity and confidentiality. The refining machine does not modify the state representation (machine variables) of the refined machine to remove the non-determinism of the

precise procedures to protect the revelation integrity and confidentiality. The design decision to employ the DSA (Digital Signature Algorithm) as the signature algorithm has been made to achieve the revelation integrity requirements since the DSA is a standard algorithm proposed by the National Institute of Standards and Technology (NIST) in 1991. For the revelation confidentiality, it was decided to employ DES (Data Encryption Standard) since the DES algorithm is has been developed and endorsed by the U.S. government as an official encryption standard in 1977. For the mailbox access control, each spy is assigned a password for the mailbox in which he/she is subscripted. The refinement mechanism in B allows for documenting design decisions at each refinement step and this links these design decisions to the relevant requirements. For example, the employment of the DSA algorithm for digital signature links to the requirements of revelation integrity through the operations signRevelation and verifyRevelation that carry out the algorithm. The KAOS goal graph provides traceability links between operations carried out in design and the requirements they achieve.

The design decisions made in the second refinement step are highlighted in the following B code:

```
REFINEMENT SpyNetworkRR
REFINES SpyNetworkR
SEES StrTokenType, Bool_TYPE, SpyNetworkUtilities

OPERATIONS

encyptedRevelation <-- encryptRevelation(revelation, identity) =
    VAR publicKey IN
    IF (spiesr~(identity): dom(spiesr)) THEN
      publicKey := spyPublicKeyr(identity);
      encyptedRevelation := SpyNetworkUtilities.DESencrypt (revelation, publicKey);
    ELSE
      encyptedRevelation := EmptyStringToken;
    END
END;

revelation <-- decryptRevelation(ecryptedRevelation, identity) =
    VAR privateKey IN
    IF (spiesr~(identity): dom(spiesr)) THEN
```

```
        privateKey := spyPrivateKeyr(identity);
            revelation := SpyNetworkUtilities.DESdecrypt (revelation, privateKey);
      ELSE
            revelation := EmptyStringToken;
      END
  END;

  signature <-- signRevelation(revelation, identity) =
    VAR privateKey, publicKey IN
    IF (spiesr~(identity) : dom(spiesr)) THEN
      privateKey := spyPrivateKeyr(identity);
      publicKey := spyPublicKeyr(identity);
      signature := SpyNetworkUtilities.DSAsign
                  (revelation, privateKey, publicKey) ;
  ELSE
    signature:= EmptyStringToken;
  END
END;

verified <-- verifyRevelation(revelation, identity, signature) =
  VAR privateKey, publicKey IN
  IF (spiesr~(identity) : dom(spiesr)) THEN
        privateKey := spyPrivateKeyr(identity);
    publicKey := spyPublicKeyr(identity);
        verified := SpyNetworkUtilities.DSAverify
                  (revelation, privateKey, publicKey, signature) ;
  ELSE
    verified := FALSE;
  END
END;

authenticated <-- certifySpy(identity, publicKey) =
  VAR key IN
 authenticated := bool(spyPublicKey(identity) = publicKey)
END;

accessAllowed <-- accessMailbox(identity, password) =
 IF (spiesr~(identity) : dom(spiesr)) THEN
  IF (password == spyMailboxPasswordr(identity)) THEN
      accessAllowed := TRUE
```

```
    ELSE accessAllowed:= FALSE
    END
    ELSE accessAllowed:= FALSE
    END
END;
```

The security algorithms used in the second refinement step are encapsulated in another machine SpyNetworkUtilities that encapsulates generic security utilities. This makes the design more modular by dividing the tasks among multiple machines, each of which provides interfaces for its operations to be used by other machines. For example, the second refinement uses the DESencrypt, DESdecrypt, DSAsign and DSAverify from the SpyNetworkUtilities machine. Separating these operations in separate machine allows for changing the security algorithms used for encryption/decryption and digital signatures seamlessly as far as the spy network machine itself is concerned since the change will be localized only in the SpyNetworkUtilities machine. The spy network machine will not be affected by the change since it remains using the same operations with the same interface while the SpyNetworkUtilities machine changes the implementation. The last refinement step derives implementation specifications for the security requirements of the spy network system. The implementation step can be done only once for each development with some constraints such as that the implementation machine has no state and cannot use abstract substitutions like non-determinism choice and parallel composition. The implementation machine imports the SpyNetworkUtilities, privateKeyArray, publicKeyArray, and spyMailboxArray machines that are then considered under the control of the implementation machine. Further, the implementation machine has to instantiate the parameters of the imported machines. The implementation machine has no variables section meaning that it has no state itself; rather it maintains the state of the imported machines. The invariant section of the implementation machine contains liking invariants between the imported state variables namely spiesArray, privateKeyArray, publicKeyArray, spyMailboxPasswordArray, and spyMailboxArray and the variables of the SpyNetworkR1 that this implementation refines. Operations in the implementation machine need to convert mathematical specifications to a format that could be translated to a programming language. For example, all the operations in the implementation machine use a loop to locate the

spy whose identity is specified in the operation parameters in the array of spies. This search in the array using a loop could be directly translated to equivalent constructs in the generated C code. The B code for the implementation specifications is as follows:

```
IMPLEMENTATION SpyNetworkI
REFINES SpyNetworkR1
USES StrTokenType, Bool_TYPE
IMPORTS
SpyNetworkUtilities, privateKeyArray(maxSpies), publicKeyArray(maxSpies),
spyMailboxArray(maxSpies)
INVARIANT
spiesArray = spiesr & privateKeyArray = spyPrivateKeyr &
publicKeyArray = spyPublicKeyr & spyMailboxPasswordArray =
spyMailboxPasswordr &
spyMailboxArray = spyMailboxPasswordr

OPERATIONS

encyptedRevelation <-- encryptRevelation(revelation, identity) =
  VAR ii, publicKey IN
  ii := 0;
  WHILE ii <= maxSpies
  DO ii := ii + 1;
  IF (spiesArray(ii) == identity) THEN
        publicKey := publicKeyArray(ii);
        encyptedRevelation := SpyNetworkUtilities.DESencrypt (revelation,
publicKey);
   ELSE
     encyptedRevelation:= EmptyStringToken;
   END
   INVARIANT
     ii : NATURAL1 & ii <= maxSpies & spiesArray = spiesr & publicKeyArray =
spyPublicKeyr
   VARIANT
     maxSpies - ii
   END
END;

revelation <-- decryptRevelation(ecryptedRevelation, identity) =
```

```
      VAR ii, privateKey IN
      ii := 0;
      WHILE ii <= maxSpies
      DO ii := ii + 1;
      IF (spiesArray(ii) == identity) THEN
            privateKey := privateKeyArray(ii);
            revelation := SpyNetworkUtilities.DESdecrypt (revelation, privateKey);
      ELSE
        revelation := EmptyStringToken;
      END
      INVARIANT
        ii : NATURAL1 & ii <= maxSpies & spiesArray = spiesr &
        privateKeyArray = spyPrivateKeyr
      VARIANT
        maxSpies - ii
      END
    END;

    signature <-- signRevelation(revelation, identity) =
      VAR ii, privateKey, publicKey IN
      ii := 0;
      WHILE ii <= maxSpies
      DO ii := ii + 1;
      IF (spiesArray(ii) == identity) THEN
            privateKey := privateKeyArray(ii);
            publicKey := publicKeyArray(ii);
            signature := SpyNetworkUtilities.DSAsign
                  (revelation, privateKey, publicKey);
    ELSE
       signature:= EmptyStringToken;
    END
    INVARIANT
      ii : NATURAL1 & ii <= maxSpies & spiesArray = spiesr &
      privateKeyArray = spyPrivateKeyr & publicKeyArray = spyPublicKeyr

    VARIANT
      maxSpies - ii
    END
  END;
```

```
verified <-- verifyRevelation(revelation, identity, signature) =
  VAR ii, privateKey, publicKey IN
  ii := 0;
  WHILE ii <= maxSpies
  DO ii := ii + 1;
  IF (spiesArray(ii) == identity) THEN
          privateKey := privateKeyArray(ii);
          publicKey := publicKeyArray(ii);
          verified := SpyNetworkUtilities.DSAverify
                  (revelation, privateKey, publicKey, signature);
   ELSE
      verified := FALSE;
   END
   INVARIANT
    ii : NATURAL1 & ii <= maxSpies & spiesArray = spiesr &
    privateKeyArray = spyPrivateKeyr & publicKeyArray = spyPublicKeyr
   VARIANT
    maxSpies - ii
   END
END;
```

```
authenticated <-- certifySpy(identity, publicKey) =
  VAR ii, pk IN
  ii := 0;
  WHILE ii <= maxSpies
  DO ii := ii + 1;
  IF (spiesArray(ii) == identity) THEN
          pk := publicKeyArray(ii);
          IF (pk == publicKey) THEN
             authenticated := TRUE;
          ELSE
             authenticated := FALSE;
          END
  ELSE authenticated:= FALSE
  END
  INVARIANT
    ii : NATURAL1 & ii <= maxSpies & spiesArray = spiesr  &
    publicKeyArray = spyPublicKeyr
  VARIANT
    maxSpies - ii
  END
```

```
accessAllowed <-- accessMailbox(identity, password) =
  VAR ii IN
  ii := 0;
  WHILE ii <= maxSpies
  DO ii := ii + 1;
  IF (spiesArray(ii) == identity) THEN
        IF (password == spyMailboxArray(ii)) THEN
          accessAllowed := TRUE
    ELSE accessAllowed:= FALSE
    END
  ELSE accessAllowed:= FALSE
  END
  INVARIANT
    ii : NATURAL1 & ii <= maxSpies & spiesArray = spiesr &
    spyMailboxArray = spyMailboxPasswordr
  VARIANT
    maxSpies - ii
  END
END;
```

The final stage is to generate C code from the implementation machine. The programming language choice is based on the programming languages available in the B tool being used. Almost all the commercial B tools generate code in C and very few of them generate ADA. The security properties should be maintained by the design decisions and the semantics of the B machines rather than by specific security construct in the programming language to which the implementation machine is translated.

Acceptance Testing

The Goal Graph Analyzer tool generates a suite of acceptance test cases derived directly from the KAOS goal graph. The tool parses the graph using a DFS algorithm with backtracking facility in order to

generate scenarios with a sequence of operation calls from the goal graph. The derived B implementation specifications are then verified against the acceptance test cases to check for compliance between the requirements model and the derived implementation and to ensure the preservation of the security properties specified in the requirement model. Test results can be used to identify areas of inconsistencies and errors either in the requirements model itself or in the B refinement steps. The acceptance test cases are considered as substantial contribution of FADSE since it strengthens the approach with extra verification of development correctness and compliance between the software and its requirements from the security standpoint. The implication of this contribution is that it increases confidence of the software security developed with FADSE. The generated test cases might be augmented with some messages and assertions for better usability of the test results. The Goal Graph Analyzer has generated the following test cases for the spy network security requirements:

```
    public static boolean testSendRevelation(String revelation, String senderId, String
recepientId)  {
            String encryptedRevelation : =
     SpyNetwork.encryptRevelation(revelation, senlderId);
            String signedRevelation : =
    SpyNetwork.signRevelation(encryptedRevelation, Senderld);
            SpyNetwork.sendRevelation(signedRevelation, recepientId)
            return true;
    public static boolean testReceiveRevelation(String revelation, String senderId,
            (String recepientId) {
                        String pk : = SpyNetwork.getPublicKey(Senderld);
                boolean certified : = SpyNework.certifySpy(senderld, pk);
                if (certified) {
                    boolean verified : =
    SpyNetwork.verifyRevelationfrevelation, senderldji;
            if (verified) {
                String decryptedRevelation : =
                SpyNetwork.decryptRevelation(revelation, recepientId);
                            return true;                    }            }
    return false;
public static boolean accessMailbox(String spyld, String password)
{   return SpyNetwork.accessMailbox(spyld, password);   }
```

In the above generated test cases, each test case describes the sequence of calls to the security operations in order to secure the

communication of revelations. The first test case outlines the scenario for the sending operation in which the encryptRevelation and signRevelation operations are called before sending the revelation. The second test case handles the scenario for the receiving operation in which the certifySpy is called to authenticate the sender spy followed by a verification of the revelation signature through calling verifyRevelation and at the end the revelation is decrypted by calling the decrptRevelation operation. The third test case tests the eligibility of access to a mailbox that could be used when spies login to their mailboxes and it calls the accessMailBox opreation. It can be observed that all the security operations have been called in the generated test cases using the main scenarios in which these operations are called. This raises the probability of error-detection when the derived implementation is verified against the acceptance test cases increasing the confidence in the security properties of the final product. Further, the coverage of the acceptance test cases provides a means to the customer to verify that the final product meets his security requirements.

Security Specifications Changes

Maintenance activities are classified into four categories according to [1, 11]: adaptive (changes in the software environment), perfective (new user requirements), corrective (fixing errors), and preventive (prevent future problems). An example of a corrective change has been chosen since corrective changes consume 21% of change requests [11]. A defective scenario threatening revelation confidentiality is as follows: A spy leaves his team and gets reallocated to another team after a message has been sent. This scenario is not handled by the current encryption/decryption solution used to protect the confidentiality of revelations since the leaving spy would receive a revelation that he is no longer eligible to receive. To correct this defect, the KAOS framework provides a conflict construct that allows the expression of situations that contradict with system requirements. Let us consider the following conflict to the RevelationConfidentiality goal.

Conflict KnowingAfterLeavingTeam

InformalDef A spy knows a revelation targeted to a team although he has left
FormalDef ◊ ∃ sp1, sp2 : Spy, te1, te2 : Team, re : Revelation
Member(sp1, te1) ∧ Member(sp2, te1) ∧ ¬ [Knows(sp2, re.Content)
$\mathcal{U}$ ❑ Member(sp2, te2) ∧ te1 ≠ te2]

This conflict could be resolved using one of the patterns for conflict resolution [4] by introducing a new goal to anticipate the conflict:

Goal *Achieve* [OneDayNoticeBeforeReallocation]
InformalDef A team relay is notified one day before a spy in his team is reallocated
FormalDef ∀ sp1, sp2 : Spy, te1, te2 : Team
Member(sp1, te1) ∧ $\Diamond_{<24h}$ Member(sp1, te2) ∧ te1 ≠ te2 ∧ Relay(sp2,te1)
⇒ ◊ (∃ mn : MemberNotif) Sending(- , mn, sp2)
∧ mn.Name = sp1 ∧ mn.Team = te2

This goal could be assigned to a reliable agent such as the big boss. Analyzing the impact of introducing the new goal shows that the RevelationConfidentiality requirement would be affected by this change. The traceability information provided by the hierarchical structure of the goal graph and the KAOS refinement mechanism direct the change impact analysis to revisit the AND refinement of the RevelationConfidentiality goal. The new goal needs to be added as a subgoal to the refinement of the RevelationConfidentiality goal. The goal graph for the RevelationConfidentiality goal would be modified as in Figure 1.6 to add the new goal:

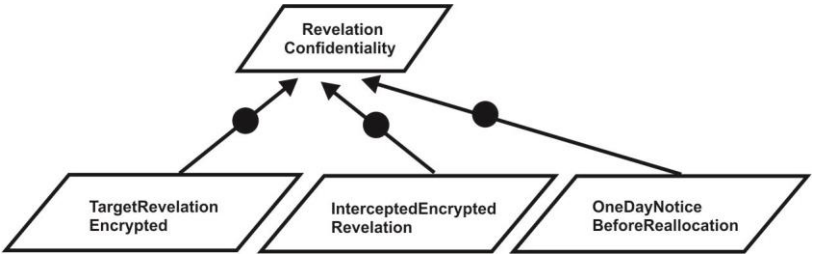

Figure 1.6. Accommodating the new goal

Since the new goal is a leaf goal, it will be operationalized using the following operation:

Operation NotifyRelayWithReallocation
Input Spy{arg relay}, Spy{arg leavingSpy}
DomPre ¬(∃ relay, leavingSpy:Spy) Notified(relay, leavingSpy)
DomPost(∃relay, leavingSpy:Spy) Notified(relay, leavingSpy)

ReqPreFor (∃ team1, team2:Team) Member(relay, team1) ^ Member(leavingSpy, team1) ^ Member(leavingSpy, team2)

This operation needs to be transformed to B in order to propagate this corrective change to the derived design and implementation. According to the change impact analysis performed with respect to the transformation of the new operation to B, we discovered that the state representation (Variables) of the SpyNetwork machine needs to be complemented with the following variables: team, relaySpies, authorizedReceiversFrom, authorizedSendersTo. These variables represent the set of assigned relays of all teams as well as the set of spies authorized to send to or receive from relays. Constraints on these variables need to be added to the invariant of the machine as follows:

team : spies-->STRTOKEN &
relaySpies <: spyId &
authorizedReceiversFrom : relaySpies >+> spies &
authorizedSendersTo : spies +> relaySpies

The definition of the operation notifyRelayWithReallocation is given below and it should be refined for design and implementation like the rest of the operations.

```
notifyRelayWithReallocation(relayId, leavingSpyId) =
  PRE relayId : NATURAL1 & leavingSpyId : NATURAL1 &
  team (relayId) = team (leavingSpyId) THEN

  IF ((relayId : ran(spyId)) & (leavingSpyId : ran(spyId))) THEN
    authorizedReceiversFrom(relayId) := authorizedReceiversFrom(relayId)
     - {leavingSpyId} ||
    authorizedSendersTo(relayId) := authorizedSendersTo(relayId)
     - {leavingSpyId}
  END;
END
```

Tasks for laboratory work №1.

1. Each student choose different type of software.

2. According to chosen software elaborate the security requirements to specified system (Integrity Goals, Confidentiality Goals, Authentication Goals, Availability Goals, Access Control Goals);
3. Make the analysis and resolution of obstacles and conflicts for specified security goals.
4. Construct model of the system with KAOS.
5. Transform built model into B.
6. Make the refinement to derive implementation specifications in B.

Requirements to the report

The report should consists of:

- title sheet;
- the aim and the task of the laboratory work;
- defined security requirements to the system;
- graphical model of the system;
- presentation oh the built model in B;
- presentation of the refinement to derive implementation specifications in B;
- results and conclusions.

Advancement questions

1. Why does some refinements are not complete?
2. What we should do to formally express these security goals?
3. What we need to do for applying the generic security model to a system?
4. In what language do we are able to expressed security goals?
5. What does the confidentiality of revelations means?
6. What does the Confidentiality of mailboxes means?
7. What authentication means in the context of the spy network case study?
8. Is it possible to elaborate the security requirements to Access Control Goals, and how?
9. How to make the analysis and resolution of obstacles and conflicts for specified security goals?
10. How to construct model of the system with KAOS? How to transform built model into B?

What we should do to get the refinement to derive implementation specifications in B?

1.2 Laboratory work 2. Applying formal methods to a certifiably secure software system

The aim and the task of the laboratory work

The aim of this laboratory work is to gain knowledge and acquire skills in the verifying security down to the source code level.

Task of the work:

- build a well-defined security property;
- build the minimal state machine model needed to prove that the model satisfies the property;
- using a mechanical verifier, prove that the security model satisfies the property;
- annotate the code with preconditions and postconditions and partition it into Event, Trusted, and Other Code;
- demonstrate conformance of the Event Code and the code preconditions and postconditions with the internal events and preconditions and postconditions of the TLS ;
- show that the Trusted Code and the Other Code are benign;
- develop tools for validating and constructing preconditions and postconditions from the source code, including the C code,
- develop tools for automatically generating test cases that check C code annotations;
- develop tools for showing conformance of annotated code with a TLS, and automatically constructing efficient provably correct code from specifications.

Preparation for laboratory work

- to clarify the aims and objectives;
- to study theoretical material given in the description.

Theoretical material

Introduction

A challenging problem therefore is how to make the verification of security-critical code affordable. Let us consider an practical approach to verifying the security of software that significantly reduces the cost of verification. This approach is formulated to support a Common Criteria evaluation of the security of a software- based embedded device called ED (Embedded Device). Satisfying the Common Criteria required a formal proof of correspondence between a formal specification of ED's security functions and its required security properties *and* a demonstration that ED's implementation satisfied the formal specification. ED, which processes data stored in different partitions of its memory, must enforce a critical security property called *data separation* to ensure, for example, that data in one memory partition neither influences nor is influenced by data in another partition. To guarantee that data separation is not violated (or, if it is violated, an exception occurs), ED relies on a separation kernel [12,13], a tamper-proof nonbypassable program mediating every access to memory.

The main is to provide evidence to the certifying authority that the ED separation kernel enforces data separation. The kernel code, which contains on the order of 3,000 lines of C and assembly code, is annotated with preconditions and postconditions in the style of Hoare and Floyd. To provide evidence that ED enforces data separation, a Top-Level Specification (TLS) of the separation-relevant behavior of the kernel, a formal statement of data separation, and a mechanized formal proof that the TLS satisfies data separation are produced. Then, the annotated code is partitioned into three categories, each requiring a different proof strategy. Finally, the formal correspondence between the annotated code and the TLS was established. Five artifacts—the TLS, the formal statement of data separation, proofs that the TLS satisfies data separation, the organization of the annotated code into the three categories, and the documents showing correspondence of the code to the TLS— were presented, along with the annotated code, as evidence supporting the certification of ED.

Let us consider the process that produces the evidence for the Common Criteria evaluation, and describes the artifacts developed during the process, and presents the formal argument justifying the approach to establishing conformance of the code with the TLS. Also,

subsection describes a technique for partitioning the code into three different categories and for reasoning about the security of each category, which reduces the cost of verification. Also, it describes an method for demonstrating the security of code. Although the method combines a number of well-known techniques for specifying and reasoning about security (for example, a state machine model, an access control matrix [14], mechanized reasoning using PVS [15], and a demonstration of correspondence between the TLS and the annotated code). Described techniques for partitioning the code and the method for proving the security of the code is able to prove cost-effective efforts to verify the security of software.

Background Separation Kernel

A *separation kernel* [13] mimics the separation of a system into a set of independent virtual machines by dividing the memory into partitions and restricting the information flow between those partitions. Separation kernels are being developed by commercial companies such as Wind River Systems, Green Hills Software, and LynuxWorks for military applications requiring Multiple Independent Levels of Security (MILS).

In a MILS environment, a separation kernel acts as a reference monitor [16]: it is nonbypassable, evaluatable, always invoked, and tamper-proof.

Common Criteria

A number of international organizations established the Common Criteria to provide a single basis for evaluating the security of information technology products [16]. Associated with the Common Criteria are seven Evaluation Assurance Levels. EAL7, the highest assurance level, requires a formal specification of a product's security functions and its security model and formal proof of correspondence between the two.

Embedded Device

The device of interest, ED, processes data in an embedded system whose memory has been divided into nonoverlapping partitions. Although, at any given time, the data stored and processed by ED in one memory partition is classified at a single security level, ED may later reconfigure that partition to store and process data at a different security

level. Because it stores and processes data classified at different security levels, security violations by ED could cause significant damage. To prevent violations of data separation, for example, the “leaking" of data from one memory partition to another, the ED design uses a separation kernel to mediate access to memory. By mediating every access, the kernel ensures that every memory access is authorized and that every transfer of data from one ED memory location to another is authorized. Any attempted memory access by ED that is unauthorized will cause an exception.

Code Verification Process

Given 1) source code annotated with Floyd-Hoare preconditions and postconditions and 2) a security property of interest, the problem is how to establish that the code satisfies the property. Let us consider a five-step process for establishing the property, each step producing one of the five artifacts. The five steps of the process are listed as follows:

- Formulate a TLS of the code as a state machine model.
- Formally express the security property as a property of the state machine model. Confirm that the property is preserved under refinement.
- Translate the TLS and the property into the language of a mechanical prover and prove formally that the TLS satisfies the property.
- Given source code annotated with preconditions and postconditions, partition the code into three categories—Event, Other, and Trusted Code—based on some criterion determined by the property of interest.
- To demonstrate that the Event Code does not violate the property of interest, construct a) a mapping from the Event Code to the TLS events and from the code states to the states in the TLS and b) a mapping from the preconditions and postconditions of the TLS events to the preconditions and postconditions that annotate the corresponding Event Code. Demonstrate separately that Trusted Code and Other Code are benign. Based on these results, conclude that the code refines the TLS.

Top Level Specification

Major goals of the TLS are to provide a precise yet understandable description of the allowed security-relevant external behavior of ED's separation kernel and to make the assumptions on which the TLS is based explicit. To achieve this, the TLS of the kernel behavior is

represented in precise natural language as a state machine model by using the style of the Military Message System (MMS) security model. The advantage of precise natural language is that it enables stakeholders with differing backgrounds and objectives, that is, the project manager, software developers, evaluators, and the formal methods team, to communicate precisely about the required kernel behavior and helps ensure, early in the verification process, that misunderstandings are weeded out and issues are resolved. Another goal of the TLS is to provide a formal context and precise vocabulary for defining data separation.

Like the secure MMS model, the state machine representing the kernel behavior is defined in terms of an input alphabet, a set of states, an initial state, and a transform relation describing the allowed state transitions. The input alphabet contains internal and external events, where an internal event can cause the kernel to invoke some process, and an external event is performed by an external host. The transform (also called the next-state relation) is defined on triples consisting of an event in the input alphabet, the current state, and the new state. Let us consider the excerpts from the TLS. To provide intuition about the observable kernel behavior of ED, it also describes the five internal events and the single external event (the last event), listed in the leftmost column of Table 2.1.

Table 2.1. Excerps from the Nonull Portion of Access Control Matrix for Partition i, $1 \leq i \leq n$

Event e in H	Memory Areas in M				
	B_i^1	D_i^1	D_i^2	…	G
Begin_Partition_i	-	-	-	-	-
Copy_B1In_D1In_i	R	W	-	-	-
Clear_D1_i	-	W	-	-	-
End_Partition_i	-	-	-	-	-
Other_NonPartProc	-	-	-	-	RW
…	…	…	…	…	…
ExtEv_B1In_i	RW	-	-	-	-

Partitions, state variables, events, and states. We assume the existence of $n > 1$ dedicated memory partitions and a single shared memory area. We also assume the existence of the following sets:

- V is a union of types, where each type is a nonempty set of values.
- R is a set of state variable names. For all r in R, TY(r) ⊆ V is the set of possible values of state variable r.
- M is a union of N nonoverlapping memory areas, each represented by a state variable.
- H = P U E is a set of M events, where each event is either an internal event in P or an external event in E.

A system state is a function mapping each state variable name r in R to a value. Formally, for all r ∈ R, s(r) ∈ TY(r). Given state s and state variable r, we abbreviate s(r) by rs.

Memory areas. The N memory areas contain N — 1 MAIs, where N — 1 = mn and m is the number of MAIs per partition. Informally, a MAI is a memory area containing data whose leakage would violate data separation. The m MAIs for a partition i, $1 \leq i \leq n$, include partition i's input and output buffers and k data areas where data in partition i are stored and processed. The Nth memory area, called G, is the single shared memory area and contains all programs and data not residing in any MAI. The set M of all memory areas is defined as the union A U{G}, where $A = \{A_{i,j} \mid 1 \leq i \leq n \wedge 1 \leq j \leq m\}$ contains the mn MAIs. For all i, $1 \leq i \leq n$, $A_i = \{A_{i,j} \mid 1 \leq j \leq m\}$ is the set of memory areas for partition i. To ensure that they are nonoverlapping, the memory areas of M are required to be pairwise disjoint.

State variables. The set of state variables contained in R are

- a partition id c,
- the N memory areas in M, and
- a set of n sanitization vectors W[1],..., W[n], each vector containing k elements.

The partition id c is 0 if no data processing in any partition is in progress and it is i, $1 \leq i \leq n$, if data processing is in progress in partition i. (Data processing can occur in only one partition at a time.) For $1 \leq j \leq k$, the Boolean value of the jth element Wj [i] of the sanitization vector for partition i is true initially and if the j-th memory area of the ith partition has been sanitized since it was last written, and otherwise false. A sanitized memory area is modeled as having the value 0.

Events. The set of internal events P ⊂ H is the union of n sets, P1,, Pn, of partition events, one set for each partition i, and a singleton set Q. Thus, P is defined by $P = [U_{i=1}^{n} P_i] \cup Q$. Processing occurs on partition

i when a sequence of events from Pi is processed. The first four events listed in Table 1 are partition events in some Pi. The first event, Begin_Partition_i, initiates data processing in partition i. The next two events process data stored in i's memory areas: Event Copy_B1In_D1_i copies data from Bl, which is an input buffer assigned to i, into a memory area Dl of i and event Clear_D1_i sanitizes memory area Dl. The event End_Partition_i concludes data processing in partition i. Q's sole member is Other_NonPartProc, which is the fifth event listed in Table 1, an abstract event representing all internal events that invoke data processing in the shared memory area G. An example is the event that copies a shared algorithm, written by some external host into a shared input buffer, to some other part of G.

The set of external events E ⊂ H is defined by $E = E^{In} \cup E^{Out} \cup$ {Ext_Ev_Other}, where $E^{In} = U_{i=1}^{n} E_i^{In}$ *and* $E^{Out} = U_{i=1}^{n} E_i^{Out}$. E_i^{In} is the set of external events writing into or clearing the input buffers of partition i and E_i^{Out} is the set of external events reading from or clearing the output buffers of partition i. The event Ext_Ev_Other represents all other external events. ExtEv_B1In_i, the last event listed in Table 1, is an example of an external event in EIn which occurs when an external host writes data (to be processed in partition i) into the input buffer B_j^1.

Partition and nonpartition functions. Operations on data in partition i, for example, an operation copying data from one MAI in partition i to another MAI in i, are called partition functions. For all i, $1 \leq i \leq n$, and, for each internal event e in Pi, there exists a partition function re associated with e. For all e ∈ Pi, Γe has the signature Γe : TY(a1) → TY(a2), where a1 and a2 are MAIs in Ai. Thus, each function Γe, where e is an internal event in Pi, takes a single argument, that is, the value stored in some MAI a1 and uses that argument to compute a value to be stored in MAI a2 as the result of event e. A nonpartition function Γe has access to data in G only.

Access control matrix. Associated with the M events and N memory areas is an M by N access control matrix AM, which indicates the access privileges that each internal event e in P (and its associated process) and each external event e in H has for each memory area a in M. The access privileges are either null for no access, R for read access, W for write access, or RW for both read and write access. Table 1 shows excerpts from the access control matrix AM. The leftmost column of Table 1 lists

the events in H and the headings of the remaining columns list memory areas in M. The rightmost column heading contains G, the only non-MAI, while the remaining column headings contain all MAIs for partition i. For all i, $1 \leq i \leq n$, AM shows the access privileges that each internal and external event has for each of i's memory areas and for memory area G. In Table 2.1, "-" denotes null access. For all i,j, $1 \leq i, j \leq n$, $i = j$, the access privilege that an event associated with i has to a memory area associated with j (not shown in Table 2.1) is null . Similarly, the access privilege that an event associated with j (not shown in Table 2.1) has to a memory area associated with i is also null.

To illustrate how AM limits access to the memory areas in M, we consider the event in the second row of Table 2.1, that is, e=Copy_B1In_D1In_i. Table 1 shows that a process invoked by e has read access to B_i^1, one of i's input buffers, and write access to D_i^1, one of i's data areas, and null access to all other memory areas in M. Thus, for event e, AM[e, B_i^1] = R, AM[e, D_i^1] = W, and AM[e, a] = null for all a, a $\in$ M, a $\notin \{B_i^1, D_i^1\}$. Similarly, the event Clear_D1_i can only write to D1 and the abstract event Other_NonPartProc only has read and write access to G. The events that begin and end data processing on i, Begin_Partition_i and End_Partition_i, cannot write to any memory area. Finally, the external event ExtEv_B1In_i invokes a process that can only read and write into the input buffer B_i^1.

System. A system is a state machine whose transitions from one state to the next are triggered by events. Formally, a system Σ is a 4-tuple Σ = (H, S, so, T), where

- H is the set of events,
- S is the set of states,
- so is the initial state,
- T is the system transform, a partial function from H x S into S. T is partial because not all events are "enabled" to be executed in the current state.

Initial state. In the initial state so, the partition id c is 0; for all i, $1 \leq i \leq n$, the MAIs in Aj are 0; and each element of the sanitization vectors W[1]... W[n] is true. Hence, in the initial state, no processing in any partition is authorized, only a nonpartition process is authorized to execute, all MAIs are zero, and all data areas are known to be sanitized.

System transform. The transform T is defined in terms of a set R of transform rules R = {Re | e ∈ H}, where each transform rule Re describes how an event e transforms a current state into a new state. The number of rules is M, one rule for each of the M events in H. No rule requires access privileges other than those defined by the access control matrix AM. The notation s and s' represents the current state and the new state, respectively. When an internal or external event e does not affect the value of any state variable r, when the precondition is not satisfied, or when the event e is not enabled, the value of r does not change from state s to state s' and the state variable r retains its current value, that is, rs = rs'
.

To denote that no state variable changes, except those explicitly named, we write $NOC_{\hat{R}}$(NO Change, except to variables in $\hat{R}$), where $\hat{R}$ ⊂ R. This notation also covers the case where the ith element of a sanitization vector changes, but no other vector elements change. For example, the postcondition rs' = x ∧ $NOC_{\{r\}}$ where x ∈ TY(r), is equivalent to rs' = x ∧ ∀r̂ ∈ R, r̂= r: $\hat{r}_{s\prime}$= r̂s.

Suppose that s is a state in S, e is an event in H, and R is the set of state variables. Let pree be a state predicate associated with e such that pree evaluates to true if e has the potential to occur in state s and false otherwise. In addition, let poste be a predicate associated with e such that poste(s, s') holds whenever e occurs in state s and s' is a possible poststate of s when event e occurs in state s. Formally, the transform rule Re in R is defined by

Re : pree(s) ⇒ poste(s; s').

Whenever the result state of every event e is deterministic (which is true in the TLS for ED), the assertion poste(s, s')

defines the poststate s' = T(e, s). To make T total on H x S, the complete definition of T is written as

$$T(e,s) = \begin{cases} s', if\ pre_e(s), where\ post_e(s,s'); \\ s, otherwise. \end{cases}$$

In the above definition, pree (s) is not satisfied implies that e has no effect, that is, essentially e does not occur. Abstractly, this models raising an exception and halting.

Examples of transform rules. For all i, 1 ≤ i ≤ n, the transform rule for e = Begin_Partition_i, which begins data processing on i, is denoted RBegin_partiti0n_i. A precondition for event e is that the partition id is 0

(that is, the system is not currently processing data on any partition) and the postcondition for e is that the partition id is i. For all i, 1 ≤ i ≤ n, and, for all states s and s', the rule Re for e = Begin_Partition_i is defined by

$R_{Begon\_Partition\_1}: c_s = 0 \Rightarrow c_{s\prime} = i \wedge NPC_{\{c\}}$.

The notation NOC{c} means that no state variable other than the partition id c can change. Similarly, for all i, 1 ≤ i ≤ n, the rule Re for e = End_Partition_i, which ends data processing on i, is defined by

$R_{End\_Partition\_i}: c_s = i \wedge \forall\, 1 \le j \le k, W_s^j[i] = true \Rightarrow c_{s'} = 0 \wedge NOC_{\{c\}}$

The expression "∀ 1 ≤ j ≤ k, W_s^j[i] - true" in the above rule means that each element of the sanitization vector for i must be true for data processing on i to end. This can be achieved by invoking clear events such as Clear_Dl_i prior to invoking End_Partition_i. The purpose of this precondition is to ensure that all data areas of partition i are sanitized prior to processing on G, on partition j, j - i, or on a new configuration of i. The transform rules RBegin_Partition_i and REnd_Partition_i are the only rules that change the value of the partition id c. Together, these rules constrain the partition id c to change from 0 to nonzero or from nonzero to 0.

Processing on a partition i can include copying data from an input buffer of partition i to a data area of partition i. Consider again the internal event e = Copy_BlIn_DlIn_i, whose transform rule is denoted RCopy_BlIn_DlIn_i . The preconditions for e are:

- The partition id c is equal to i.
- The invoked process must have read access R for partition i's Input Buffer 1 and write access W for Data Area 1 in partition i.
- Postconditions for e are:
- The element for Data Area 1 in partition i's sanitization vector becomes false (because the event stores the value of Buffer 1 in Data Area 1).
- A function of the value in partition i's Input Buffer 1 is written into partition i's Data Area 1.
- No other state variable changes.

For all i, the rule Re for event e = Copy_BlIn_DlIn_i is defined by

$R_{Copy\_B1In\_D1In_i}: c_s = i \wedge$ (1)

$AM[e, B_j^i] = R \wedge AM[e, D_j^i] = W$ (2)

$\Rightarrow W_{s'}^1[i] = fasle \wedge$ (3)

$D^{i}_{i,s\prime} = \Gamma_e\left(B^{1}_{i,s}\right) \wedge$ (4)

$NOC_{\{W^1[i],D^1_i\}}.$ (5)

As the fourth and final example of a transform rule, consider the rule for the internal event e = Other_NonPartProc, which represents all nonpartition processing events. The precondition is that the partition id c is 0 (that is, the system is not currently processing data on any partition). The effect is that some part of memory area G may change. The rule Re for e = Other_NonPartProc is defined by

$$R_{Other\_NonPartProc}: c_s = 0 \wedge AM[e, G] = RW$$
$$\Rightarrow G_{s\prime} = \Gamma_e(G_s) \wedge$$
$$\forall r \in R, r \neq G: r_{s'} = r_s.$$

Security Property: Data Separation

To operate securely, ED must enforce data separation, that is, it must prevent insecure data flows. Informally, this means that ED must prevent data in a partition i from influencing or being influenced by 1) data in a partition j, where i ≠ j; 2) data in an earlier configuration of partition i; or 3) data stored in G. To demonstrate that the TLS enforces data separation, it is proved that it satisfies five subproperties, namely, No-Exfiltration, No-Infiltration, Temporal Separation, Separation of Control, and Kernel Integrity.

No-ExfUtration Property

The No-Exfiltration Property states that data processing in any partition j cannot influence data stored outside the partition. This property is defined in terms of the set Aj (the MAIs of partition j); the entire memory M; the internal events in Pj, which invoke data processing in j; and the external events in $E_j^{In} \cup E_j^{Out}$, which affect data in j's input and output buffers.

Property 2.1 (No-Exfiltration). *Suppose that states* s *and* s' *are* in state set S, event *e* is in H, memory area a is in M, and *j* is a partition, $1 \leq j \leq n$. Suppose further that $s' = T(e, s)$. If *e* is an event in $P_j \cup E_j^{In} \cup E_j^{Out}$ and $a_s \neq a_s'$, then *a* is in A_j.

2.2.2 No-Infiltration Property

The No-Infiltration Property states that data processing in any partition i is not influenced by data outside that partition. It is defined in terms of the set Ai, which contains the MAIs of partition i.

Property 2.2 (No-Infiltration). Suppose that states $s_1, s_2, s_1', \text{and } s_2'$ are in S, event e is in H, and i is a partition, 1 ≤ i ≤ n. Suppose further that s_1' = T (e, s1) and s_2' = T (e, s2). If, for all a in Ai, as1 = as2, then, for all a in Ai, $a_{s_1'}$=$a_{s_2'}$.

Temporal Separation Property

This property ensures that no data (for example, Top Secret data) stored in the ith partition during one configuration of the partition can remain in any memory area of a later configuration (for example, processing Unclassified data) of that same partition i. The property is guaranteed if the k data areas in any partition i are clear when the system is not processing data in that partition, for example, from the end of a processing thread in one partition to the start of a new processing thread in the same or a different partition. The set of states in which the system is not processing data stored in a partition is exactly the set of states in which the partition id c is 0. This fact is used in stating the property.

Property 2.3 (Temporal Separation). For all states s in S, for all i, 1 ≤ i ≤ n, if the partition id cs is 0, then the k data areas of partition i are clear, that is, $D_{i,s}^1 = 0, \ldots, D_{i,s}^k = 0$.

Separation of Control Property

This property states that, when data processing is in progress on partition i, no data is being processed on partition j, j ≠ i, until processing on partition i terminates. The property is defined in terms of the partition id c and the set Di of k data areas in partition i, $\text{Di} = \{D_i^j \mid 1 \leq j \leq k\}$.

Property 2.4 (Separation of Control). Suppose that states s and s' are in S, event e is in H, data area a is in M, and j, where 1 ≤ j ≤ n, is a partition id. Suppose further that s' = T(e ,s). If neither cs nor cs' is j, then as = as' for all a ∈ Dj.

Kernel Integrity Property

The Kernel Integrity Property states that, when data processing is in progress on partition i, the data stored on memory area G does not change. This property is defined in terms of G and the set Pi of events for partition i.

Property 2.5 (Kernel Integrity). Suppose that states s and s' are in state set S, event e is in H, and i is a partition, $1 \leq i \leq n$. Suppose further that s' = T (e, s). If e is a partition event in Pi, then Gs' = Gs.

Formal Verification

To formally verify that the TLS enforces data separation, the natural language formulation of the TLS was translated into TAME (Timed Automata Modeling Environment) [18], a front end to the mechanical prover PVS [19] which helps a user specify and reason formally about automata models. This translation requires the completion of a template to define the initial states, state transitions, events, and other attributes of the state machine E. The TAME specification provides a machine version of the TLS that can be shown mechanically to satisfy the defined properties. After constructing the TAME specification of the TLS, we formulated two sets of TLS properties in TAME —invariant properties and other properties—which together formalize the five subproperties. Then, for each set of properties, we interactively constructed (TAME) proofs showing that the TAME specification satisfies each property. The scripts of these proofs, which are saved by PVS, can be rerun easily by the evaluators and serve as the formal proofs of data separation. One benefit of TAME is that the saved PVS proof scripts can be largely understood without rerunning them in PVS.

Partitioning the Code

To show formally that the separation kernel enforces data separation, we must prove that the kernel is a secure partial instantiation of the state machine $\sum$ defined by the TLS. The formal verification establishes formally that a strict instantiation of the TLS enforces data separation. A partial instantiation of the TLS is an implementation that contains fine-grained details which do not correspond to the state machine $\sum$ defined in the TLS. A secure partial instantiation of the TLS is a partial instantiation of the TLS in which the fine-grained details that do not correspond to the TLS are benign. Let us consider how the formal foundation for the proof that the code is a secure partial instantiation of the TLS.

The proof that the code for the ED kernel is a secure partial instantiation of the TLS is based on a demonstration that all kernel code falls into three major categories and one subcategory, with proofs that the

code in each category satisfies certain properties. The categories are given as follows:

Event Code is kernel code that implements a TLS internal event e in P and touches one or more MAIs. For each segment of Event Code, it is checked that

- the concrete translation of the precondition in the TLS for the corresponding event e is satisfied at the point in the kernel code where the execution of the event code is initiated, and
- the concrete translation of the postcondition in the TLS for the corresponding event e is satisfied at the conclusion of Event Code execution.

Trusted Code is kernel code that touches MAIs but is not Event Code. This code does not correspond to behavior defined by the TLS and may have read and write access both to MAIs and to memory areas outside the MAIs. It is validated either by a proof that the code does not permit any nonsecure information flows or, in rare instances, by external certification. The TLS makes explicit any assumptions used in connection with the Trusted Code and its behavior. The proofs for a given segment of the Trusted Code characterize the entire functional behavior of that Trusted Code by using Floyd-Hoare style assertions at the code level and show that no nonsecure information flows can result from that code.

Other Code is the kernel code that is neither Event Code nor Trusted Code. More specifically, Other Code is kernel code which does not correspond to any behavior defined by the TLS and has no access to any MAI.

A subset of the Other Code, called Validated Code, is code with no access to MAIs which is still security relevant because it performs functions necessary for the kernel to enforce data separation. These functions include setting up the MMU, establishing preconditions for the Event Code, etc. Floyd-Hoare style assertions at the code level are used to prove that Validated Code correctly implements the required functions.

The kernel code was manually partitioned into Event, Trusted, and Other Code. A first pass through the code showed that only a small number of functions could reset the MMU (that is, change the access permissions to memory areas). Apple's Xcode development tool [20] was used to search the kernel code for all calls to these functions. Each such

call was inspected to determine the memory areas to which access was granted. By analyzing the access granted to code segments categorized as Other Code, one can verify that functions called in these code segments have no access to any MAI.

Partitioning the code in this manner dramatically reduces the cost of code verification since only the Event Code, a small part of the code, needs to be checked for conformance to the TLS. In ED, Event Code and Trusted Code comprised less than 10 percent of the code. The remaining 90 percent was Other Code.

Demonstrating Code Conformance

Demonstrating that the kernel code conforms to the TLS requires the definition of two mappings. To establish correspondence between concrete states in the code and abstract states in the TLS, a function a is defined which relates concrete states to abstract states by relating concrete entities (such as memory areas, code variables, and logical variables) to abstract state variables in the TLS (such as MAIs and the partition id) and mapping the value space of each concrete entity to that of its corresponding abstract state variable. For example, a maps the actual physical addresses of the MAIs to their corresponding abstract state variables in the TLS. In the ED kernel code, a maps a global variable partitioned, corresponding to the partition id, to the TLS partition id variable c. The TLS sanitization vectors have no analogs in the code. Instead, a predicate can be inferred from the code to indicate whether a memory area is sanitized. To represent sanitization in the concrete machine, new logical variables (for example, part_data1_sanitized_i) are introduced, and a maps these variables to elements of the sanitization vectors in the TLS. The map *a* also maps the Event Code to events in the TLS. Another map Φ relates assertions at the abstract TLS level to equivalent assertions at the code level derived from the abstract assertions and the map *a*.

Using Φ to relate preconditions and postconditions for an event in the TLS to the derived preconditions and postconditions for the corresponding Event Code, we next determine, for each piece of Event Code, sets of code-level preconditions and postconditions that match the derived preconditions and postconditions as closely as possible. Fig. 1 shows the Event Code corresponding to the internal event Copy_B1In_D1In_i in the TLS and the code-level preconditions and

postconditions for this Event Code. Although the Event Code for Copy_B1In_D1In_i consists of only a single function call, generally, Event Code may consist of any block of code. In Fig. 2.1, the top box contains the preconditions, then the indented Event Code is listed, and, finally, the bottom box contains the postconditions. Each precondition and postcondition has the form {Assertion_Name : Assertion}. Generally, the match between assertions in the TLS and derived code-level assertions is not exact because auxiliary assertions are added (see Fig. 2.1) to express the correspondence between variables in the code and physical memory areas4 (for example, CopyDIn_local_datain), 2) to save values in memory areas as the values of logical variables (for example, CopyDIn_value_data), and 3) to express error conditions (for example, CopyDIn_copy_size_datain) that the TLS abstracts away via type correctness.

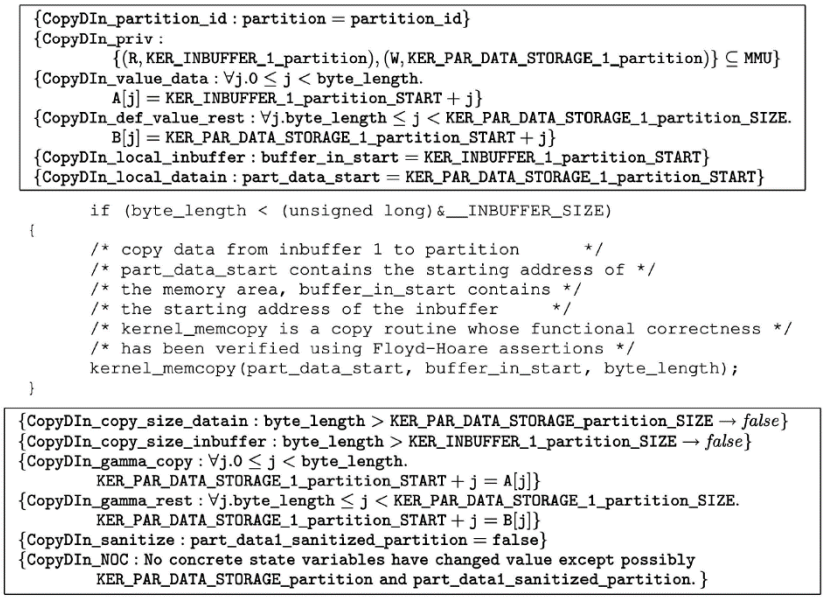

Fig. 2.1. Event Code and eent-level assertions for the event Copy_B1ln_D1ln_1

The derivation of the necessary code-level assertions is also complicated by the code itself. For example, although there is a global variable partitioned in the code, in many of the routines implementing Event Code, the partition id used in the routine is an argument that is passed into the routine. This results in a code-level precondition asserting that the local variable for the partition id is equal to the global variable partitioned (for example, CopyDIn_partitioned in Fig. 2.1).

Table 2.2. Mapping Preconditions in the Code to Preconditions in the TLS

Precondition $\Phi(pre_e)(s_c)$ Desired in the Code	Assertion in Annotated Code	Precondition $pre_e(s)$ in the TLS	Ref. No.	Description
CopyDIn_partition_id	§8.4,P5	cs = i	(1)	Partition id is i
CopyDIn_priv	§8.4,TLS1*	AM(e, B_i^1) = R AM(e, D_i^1) = W	(2)	R access for Input Buffer 1, W access for Data Area 1
CopyDIn_value_data	§8.4, P4*	$B_{i,s}^1$	-	Value of data in Input Buffer 1
CopyDIn_def_value_rest	§8.4,TLS4	$D_{i,s}^1$	-	Value of Data Area 1
CopyDIn_local_inbuffer	§8.4, TLS3*	-	-	Local variable for Input Buffer 1
CopyDIn_local_datain	§8.4,TLS2*	-	-	Local variable for Data Area 1

Table 2.3. Mapping Postconditions in the Code to Postconditions in the TLS

Postcondition $\Phi(poste)(sc, s'c)$ Desired in the Code	Assertion in Annotated Code	Postcondition poste(s,s') in the TLS	Ref No.	Description
CopyDIn_copy_size_datain	§8.4, R2*	-	-	Wrong size —> Error return
CopyDIn_copy_size_inbuffer	§8.4, R3*	-	-	Wrong size —> Error return

CopyDIn_gamma_copy	§8.4, R7*	$D^1_{i,s}$=Γ($B^1_{i,s}$)	(4)	Copy to Data Area 1
CopyDIn_gamma_rest	§8.4,TLS6	-		Rem Data Area 1 unchged
CopyDIn_sanitize	§8.4,TLS5 *	$W^1_s[i] = false$	(3)	Data Area 1 not sanitized
CopyDIn_NOC	By inspection	$NOC_{\{W^1[i],D^1_i\}}$	(5)	No other change

After defining the desired sets of code-level preconditions and postconditions, we check whether these assertions are among the assertions already proven in the annotated C code. The annotated C code often refers to memory areas by indexing into arrays that define memory maps in the code, whereas the mapping a refers to memory areas by their actual physical addresses. Thus, to be equivalent to the desired assertions, the assertions in the annotated code frequently need dereferencing. For example, the annotated C code assertion, TLS2 (see Table 2.2) is defined by

part_data_start= (unsigned char*)

ker_rtime_mmu_map[partition].part_data_start,

which sets the variable part_data_start to the starting address of the data area in the partition by indexing into the real-time memory map in the code and selecting the part_data_start member of the structure corresponding to that array element. Dereferencing the index into the array and pointer into the structure yields the memory area KER_PAR_DATA_STORAGEe_partition_START, the actual physical address of the partition data area, which stores the value used in the code-level precondition CopyDIneocal_datain (see the last line of the top box in Fig. 2.1).

In the initial attempt to match a precondition and postcondition in the annotated C code with each desired precondition and postcondition, either

- the desired assertion exactly matched an assertion in the annotated code,
- the desired assertion exactly matched an assertion in the annotated code, except dereferencing was required,
- the desired assertion was a close but not exact match of an assertion in the annotated code, or

- no code assertion exactly or approximately matched the desired assertion.

Let us consider the annotated C-code to ensure that assertions corresponding to all of the desired preconditions and postconditions were added to and verified on the code. (In general, it is sufficient to include strongest postconditions implying the derived assertions.) For example, assertions about a predicate SANITIZED on memory areas were added to the annotated code to provide correspondence to the necessary code-level assertions about the sanitization of memory areas. To show correspondence between the preconditions and postconditions in the code and the TLS, two tables were created for each TLS event. Tables 2.2 and 2.3 are the correspondence tables for the preconditions and postconditions of the transform rule for the TLS event Copy_B1In_D1In. In the tables, s and s' = T(e, s) represent the abstract prestate and poststate, sc and s'c represent the concrete prestate and poststate, and $ maps abstract predicates to corresponding concrete predicates.

In Tables 2.2 and 2.3, the first column contains the label of a desired code-level precondition or postcondition from Fig. 2.1, the second column gives the location (the section number and assertion label) of the corresponding assertion in the annotated C code, the third column contains the corresponding precondition or postcondition (if any) in the TLS, the fourth column gives the reference number of the corresponding assertion in the transform rule, and the fifth column briefly describes the assertion. In cases where no corresponding assertion exists in the TLS, " appears in both the third and fourth columns. An asterisk in the second column indicates that, for equivalence between the assertion in the annotated code and the desired code assertion to hold, the assertion in the annotated code requires dereferencing.

Tables 2.2 and 2.3 show that, for every precondition and postcondition of CopyB1In_D1In_i, there is an equivalent precondition or postcondition in the annotated code. Therefore, we have shown that, for CopyB1In_D1In_i, the full code-level preconditions and postconditions imply the TLS preconditions and postconditions. Using the same techniques, we have also demonstrated the analogous result for the remaining events. The Event Code implementing the separation kernel is a refinement of the TLS.

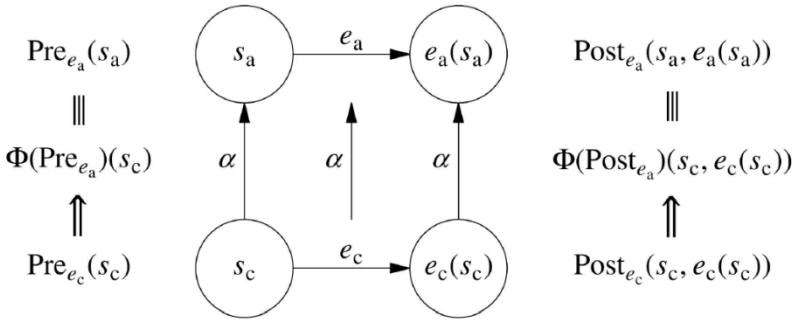

Fig. 2.2. Relations to establish between concrete and abstract transitions and preconditions and postconditions.

Formal Foundations

Let us consider the classical theory of refinement [21], a technique for proving that a concrete state machine model conforms to (that is, is a refinement of) an abstract state machine model, into a form that we can use to show that the behavior of the kernel code conforms to the behavior captured in the TLS. Also this subsection covers the formal foundation for the method of proving refinement and describes how have been applied it to verify that the kernel code correctly implements the TLS. The refinement proof technique that we use is closed under iteration.

Adapting the Classical Theory of Refinement

To begin, a function α is defined which maps each concrete state at the code level to a corresponding abstract state in the TLS state machine Σ by relating variables at the concrete code level to variables at the abstract TLS level. Variables at the concrete level include variables in the code, predicates defined on the code, logical history variables, and memory areas. Among the most important memory areas treated as concrete state variables are the data areas and the input and output buffers assigned to each partition, all of which are central to reasoning about possible information flows. Provided each possible value of a concrete state variable can be represented by some possible value of the corresponding abstract state variable (as is true for ED), the map *a* from concrete to abstract state variables induces a map $\alpha : S_c \rightarrow S_a$ from concrete to abstract states in the obvious way.6 Once α is defined at the level of states in terms of state variables and their values, the set Ec of

Event Code segments is identified, and a is extended to map each code segment ec in Ec to a corresponding internal event ea = α (ec) in the TLS.[7]

The map a from concrete states to abstract states provides a means of taking any predicate Pa : Sa → Bool on abstract states and deriving a corresponding predicate Φ(Pa) : Sc → Bool on concrete states as follows:

$$\Phi(P_a)(S_c) \triangleq P_a(\alpha(S_c)),$$

where sc is any state in Sc. Analogously, α can be used to derive a predicate Φ(Pa) : Sc x Sc → Bool on pairs of concrete states from a predicate on pairs of abstract states as follows:

$$\Phi(Pa)(s_c^1, s_c^2) \triangleq Pa(\alpha(s_c^1), \alpha(s_c^2)),$$

where s_c^1 and s_c^2 are any states in Sc. The map Φ is used to relate preconditions and postconditions in the code to preconditions and postconditions in the TLS (see Fig. 2.2). Note that preconditions (at both levels) apply only to one state. To capture the fact that an event changes only certain state variables (indicated at the abstract level by the notation NOC), the postconditions are represented at both levels as predicates on two states.

In Fig. 2.2, we follow the convention of representing α (sc) by sa. Note that, although the preconditions and postconditions on the concrete and abstract transitions in Fig. 2.2 are denoted analogously, their required relationships to their corresponding transitions differ. In particular, the precondition Preea (sa) is a guard that, when false, prevents ea from firing, while the precondition Preec (sc) is simply an assertion known to hold before ec fires. Moreover, the postcondition Postea (sa,ea(sa)) is intended to capture the effect of the action ea on the state sa, while the postcondition Postec (sc,ec (sc)) is simply an assertion known to hold for the states before and after ec fires. Hence, the requirements for the abstract preconditions and postconditions fulfill the requirements for concrete preconditions and postconditions (but not vice versa). Thus, in the refinement proof method below, an abstract TLS can play a role analogous to concrete code with respect to a still more abstract TLS.

To establish equivalence between the behavior of the kernel code and a subset of the behavior modeled in the TLS, it is sufficient to prove, in the simplest case, that, for every e_c in E_c, the following conditions hold:

- Whenever the concrete code segment ec is ready to execute in state sc, some concrete precondition Preec holds, where Preec implies

Φ(Preea), the concrete precondition derived from the abstract precondition

- for ea = a(ec).

- Whenever the concrete precondition Preec holds for the current program state sc, some concrete postcondition Postec holds for the pair of program states (sc, ec(sc)) immediately before and immediately after the execution of ec, where Postec implies Φ(Postea), the concrete postcondition derived from the abstract postcondition for ea.

- The diagram in Fig. 2.2 commutes whenever Preec (sc) holds.

- Although this method requires the proof of conditions 1, 2, and 3, it is essentially condition 3 that is needed for a to be a refinement mapping. To prove condition 3, it is normally sufficient to prove conditions 1 and 2.

Theorem 2.1. Provided $\forall s, s' \in S_a: Pre_{e_a}(s) \Rightarrow [Post_{e_a}(s, s') \equiv (s' = e_a(s))]$ conditions 1 and 2 imply condition 3.

Proof. Be hypothesis, we know that

$$\forall s, s' \in S_a: Pre_{e_a}(s) \Rightarrow [Post_{e_a}(s, s') \equiv (s' = e_a(s))]$$

And may assume that conditions 1 and 2 hold. Further, by the hypothesis of condition 3, we may also assume that

$Pre_{e_c}(s)$.

By condition 1, it follows from (ii) that $\Phi(Pre_{e_a})(s_c)$, which means, by the definition of Φ, that

$$Pre_{e_a}(\alpha(s_c)).$$

Furthermore, by condition 2, we have

$Pre_{e_c}(s_c) \Rightarrow Post_{e_c}(s_c, e_c(s_c))$,

and

$$Post_{e_c}(s_c, e_c(s_c)) \Rightarrow \Phi Post_{e_a}(s_c, e_c(s_c)).$$

Thus,

$Pre_{e_c}(s_c)$ (by (ii))

$\Rightarrow Post_{e_c}(s_c, e_c(s_c))$ (by (iv))

$\Rightarrow \Phi Post_{e_a}(s_c, e_c(s_c))$ (by (v))

$\Leftrightarrow Post_{e_a}(\alpha(s_c), \alpha(e_c(s_c)))$ (by the definition of Φ)

$\Leftrightarrow \alpha(e_c(s_c) = e_a(\alpha(s_c))$ (by (i) and (iii)).

But, the last assertion means that the diagram in Fig. 2.2 commutes, which is the conclusion of condition 3.

The hypothesis of Theorem 2.1 does not truly limit its use, provided that the abstract postcondition exactly captures all possible effects of the abstract transition. In particular, suppose that the definition of the abstract transition allows nondeterminism, and that one has established, based on conditions 1 and 2 and the hypothesis of condition 3, that $Post_{ea}(a(s_c), a(e_c(s_c)))$, that is, that $Post_{ea}(s_a, a(e_c(s_c)))$. Then, to fulfill the hypothesis of Theorem 4.1, one can simply replace e_a by its deterministic instance for which the abstract poststate $e_a(s_a)$ is $a(e_c(s_c))$.

Establishing conditions 1-3 guarantees that, whenever the code segment e**c** executes in the code, there is an enabled event e_a in the TLS that causes a transition from the abstract image s_a under a of the concrete prestate s**c** at the code level into an abstract state $e_a(s_a)$ that is the abstract image under a of the concrete poststate $e_c(s_c)$ at the code level. More concisely, conditions 1, 2, and 3 imply that there exists an abstract transition that models the concrete transition.

The relation of Event Code segments to abstract events can be slightly more complex than shown in Fig. 2.2. For example, in some cases, e_c may implement more than one event. However, these more complex cases can usually be handled similarly. When a concrete event implements n abstract events, for example, one looks for a partition $Pre_c = Pre_c^1 \oplus \ldots \oplus Pre_c^n$ of the concrete precondition $Pre_{\mathbf{c}}$ such that, when the ith part $Pre^i_{\mathbf{c}}$ holds, the code e_c implements the ith abstract event. Then, one establishes, for each i, a commutative diagram analogous to the diagram in Fig. 2.2.

The argument that the kernel code of ED ensures data separation is based on relating executions of the code to executions in the TLS. To begin, we observe that a maps ED's initial state via a to an allowed initial state in the TLS. To support the remainder of the argument, the Event Code set E**c** and the code-level map *a* are extended to cover the Other Code. Most Event Code segments consist of a single program statement. In contrast, Other Code contains many lengthy code segments which simply manipulate local variables inside a function or procedure and do not map to any abstract event. Such segments typically occur prior to an Event Code segment. We model these Other Code segments at the abstract level by a no-op ("do nothing") event implicitly included in the TLS. It is possible to map the effect of a segment of the Other Code to a no-op in the TLS because, unlike Event and Trusted Code, the Other Code has no access to MAIs. Because every code segment in the Event

or Other Code is modeled either by an abstract TLS event with concrete and abstract transitions related as in Fig. 2.2 or by a no-op in the TLS, it follows that every execution of this part of the code corresponds to an execution in the TLS.

Trusted Code in the ED kernel can be related to the TLS as follows: First, it is established that no segment of the Trusted Code causes insecure data flows. Some segments of the Trusted Code have been verified, and the remaining segments have been certified externally to cause no insecure information flows. The state change caused by each Trusted Code segment is then shown to map to the result of either a no-op in the TLS or some sequence of events in the TLS. In the overall argument that an execution of concrete code always maps to a possible execution in the TLS, each Trusted Code segment is treated as an indivisible unit. In ED, this is possible because each Trusted Code segment executes within a single partition and executions within a partition are never interrupted.

Combining this reasoning with the additional assurance that a relates concrete data and buffer memory areas to abstract ones and thus models all information flows involving MAIs, it follows that all kernel behavior relevant to data separation at the concrete level is modeled at the abstract level. Thus, the Data Separation Property proven at the abstract level also holds at the concrete level.

Uses and Proof Methods for Refinement

Although some details of how they are applied may vary, commutative diagrams are widely used to describe the required relationships between transitions at the concrete and abstract levels in a refinement relation (sometimes referred to as an *abstraction* relation).

When model checking is used to verify systems, a typical approach is to generate an abstract model automatically using data abstraction or data type reduction in a way that guarantees that the original system is a refinement of the model. Thus, any properties verified of the abstract model that are preserved under refinement will also hold for the system. In this approach, refinement is a given and need not be proved. For us, it is not feasible to use model checking to produce an abstract model. Due to the state explosion problem, model checking for verification has mostly been applied to hardware systems. Although, to some extent, methods such as abstraction refinement have made it more feasible to

apply model checking to software systems, model checking is better for detecting software bugs than for verifying software.

The concept of refinement also arises in the context of proving an implementation relation from a more concrete system model to a more abstract one. For example, the decision procedures can be used, or PVS to verify that concrete models implement specifications by proving that a set of diagrams commute, where the diagram for each transition captures the correlation between sequences of instructions at the concrete and abstract levels can be used. In order to avoid use of commutative diagrams the compositional model checking for proving implementation, in particular by model checking individual transitions separately and then proving that the results compose can be used. In the context of hierarchical verification, a diagram that relates concrete states to abstract states and concrete programs (or program fragments) to abstract transitions in which the poststate is mathematically defined in terms of the prestate can be used.

Fig. 2.2 shows required relationships, along with the commutative diagram. We also make explicit that 1) at the state variable level, the "state variables" mapped by the mapping function can be derived variables or logical variables that, for example, capture history and 2) postconditions are actually predicates on two states.

Applying Techniques to Other Security Properties

Two important classes of security properties are safety and liveness. Any property p can be expressed as the intersection of a safety property and a liveness property. Informally, a safety property states that nothing "bad" happens during execution and a liveness property states that something "good" happens during execution. A set of executions is called a *property* if membership in the set is determined by each execution alone, without reference to other executions in the set.

A security property p must be preserved under refinement. It is well known that safety properties are preserved under refinement but that liveness properties are not [12]. Hence, techniques can be used to guarantee security properties that are safety properties. It is easy to show that four of the security properties No-Exfiltration, Temporal Separation, Separation of Control, and Kernel Integrity—are safety properties. Therefore, all four properties are preserved by refinement. The fifth property, No-Infiltration, is not a safety property because it is not a

property of executions but a property of sets of executions. However, it is easily shown to be preserved under refinement.

Such approach may be applied to many applications that, like ED, enforce access control. Applications that enforce access control restrict the operations that subjects (for example, users) can perform on objects (for example, data). As long as the access control policy can be represented as a safety property, the approach applies. A second important class of applications to which the approach applies are those described by Schneider, which use Execution Monitoring (EM) to enforce security. Examples of EM mechanisms are reference monitors, firewalls, and other operating system and hardware-based enforcement mechanisms described in the literature. Excluded from this class are applications that use more information than would be available from observing only the states of a single system execution.

Schneider shows that the security properties enforced by EM mechanisms are safety properties.

Applying Method to Additional Kernel Properties

In ED's certification, the task is to develop a TLS of ED's kernel code, to verify that the TLS satisfies data separation, and, finally, to demonstrate conformance of the kernel code to the TLS. An important aspect of the approach is that, if required, we can construct a refinement of the TLS by adding new variables and events to the TLS to capture some behavior of (that is, events in) the Other Code. If the security properties that we wish to prove about this additional behavior are preserved by refinement, then we can formally state and prove the new security properties for the refinement of the TLS and show correspondence between the related portion of the Other Code and the new behavior. Because the proof method can be iterated through a series of refinements, the proof of data separation remains valid under such a refinement of the TLS.

Lessons learned. Software Design Decisions

Three software design decisions were critical in making code verification feasible. One major decision was to use a separation kernel, a single software module to mediate all memory accesses. A design that distributed the checking of memory accesses would have made the task of proving data separation much more difficult. A second critical

decision was to keep the software simple. For example, once initiated, data processing in a partition was run to completion unless an exception occurred. In addition, ED's services were limited to the essential ones: The temptation to add new services late in the development was resisted. The third critical decision was enforcing "least privilege." For example, if a process only requires read access to a memory area, the kernel only grants read, *not* read and write, access.

Top-Level Specification

One significant challenge was to understand the externally visible security-relevant behavior of the separation kernel. Both scenarios and the SCR (Software Cost Reduction) tools [26] were useful in extending understanding of the kernel behavior. To begin, we formulated several scenarios, that is, sequences of events, and specified the kernel response to those events. After specifying a state machine model of the kernel in SCR, we ran the scenarios through the SCR simulator. As expected, formulating the scenarios and running them through the simulator exposed gaps in the understanding. Both the scenarios and the questions raised are valuable in eliciting details of the security-relevant kernel behavior from ED's development team.

Once the kernel's required behavior was understood, approximately 2.5 weeks were needed to formulate the TLS and the data separation property. The complete statement of the TLS, including the assumptions, is only 15 pages long. Keeping the size of the TLS small was critical for many reasons. It simplified communication with the other stakeholders, changing the specification when the kernel behavior changed, translating the specification into TAME, and proving that the TLS enforced data separation.

During the certification process, the natural language representation of the TLS enabled stakeholders with differing backgrounds and objectives—for example, the project manager and the evaluators—to communicate easily with the formal methods team about the kernel's required behavior. Discussion among the various stakeholders helped ensure that misunderstandings were avoided and issues were resolved early in the certification process. This natural language representation of the TLS for ED contrasts with the representations used in many other formal specifications of secure systems, which are often expressed in specialized languages such as ACL2. Moreover, any ambiguity inherent

in the natural language representation was removed by translating the TLS into TAME since the state machine semantics underlying TAME is expressed as a PVS theory. One component of the TLS in particular, the access control matrix, facilitated communication between the formal methods team and other stakeholders. Although the matrix was largely redundant of other parts of the TLS, stakeholders could easily understand the matrix and thus validate constraints on the access privileges of processes invoked by each event. The matrix was also useful in identifying the events and MAIs to be included in the TLS.

Mechanized Verification

TAME's specification and proof support significantly simplified the verification effort and can require a total of about 3.5 weeks. Approximately 1.5 weeks can be required to produce the final TAME model of the TLS and to document the correspondence between the TAME model and the TLS. Some of this time was required to choose appropriate data structures for representing the state variables and the parameters of actions in TAME. The higher order nature of PVS made it feasible to handle the unspecified number of memory areas in the TLS by representing the overall memory content in TAME as a function from a set of memory areas to storable values and, in general, to produce a very compact TAME specification (368 lines long). Once the data representations is chosen, translating the TLS and the five subproperties into TAME can require at about three days. Adjusting the TAME specification to reflect later changes in the TLS can require only a few hours. To illustrate the TAME representation.

About two weeks can be needed to formally verify that the TLS enforces data separation. Most of this time can be spent formulating an efficient proof approach and then developing a new TAME strategy to implement the approach. The new PVS strategy, designed to simplify the proof guidance in the presence of the data structures used in the TAME specification, is used in the proofs of all subproperties and is subsequently proven useful in other TAME applications. Once the strategy is developed, the time required to develop the proof scripts interactively in TAME can be one day. Adding and proving a new subproperty suggested by an evaluator can require under one hour. The proof script of each subproperty can be executed in two minutes or less.

Iterating the Refinement Method

Fig.2.3 illustrates the mappings, predicates, and relationships between assertions connected with the proof of successive refinements from an automaton at level c through an automaton at level b to an automaton at level a. We wish to prove that if the analogs of conditions 1, 2, and 3 from Section 4.1 hold for the c-to-b and b-to-a relations, then conditions 1, 2, and 3 hold fort he composed c-to-a relation in which $\alpha \triangleq \alpha_1 \circ \alpha_2$ and $\Phi \triangleq \Phi_1 \circ \Phi_2$. Let us use s_b to denote α_2 (s_c), s_a to denote α_1 (s_b), and S_a, $S_{b\prime}$, and S_c to denote the sets of states at levels a, b, and c. Wefirstneed a lemma.

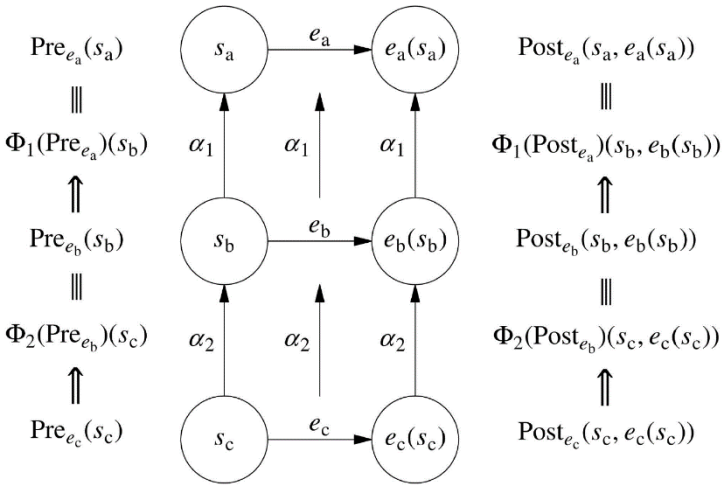

Fig. 2.3. Relationships in successive refinements

Lemma A.1 Let α: $S_c \to S_a$ and let Φ be the map from predicates on S_a to predicates on S_c incluced by α, that is such that, for any predicate P_a and any element $s_c \in S_c$, $\Phi(P_a)(s_c) \triangleq P_a(\alpha(s_c))$. If P_a and Q_a are predicates on S_a such that $P_a \Rightarrow Q_a$, then $\Phi(P_a) \Rightarrow \Phi(Q_a)$.

Proof. Suppose that P_a and Q_a are two predicates on S_a, for which $P_a \Rightarrow Q_a$. This means that, $\forall s_a \in S_a, P_a(s_a) \Rightarrow Q_a(s_a)$. Let s_c be any element of S_c. Then,

$\Phi(P_a)(s_c) = P_a\big(\alpha(s_c)\big)$ (by the definition of Φ)
$\Rightarrow Q_a(\alpha(s_c))$ (since $P_a \Rightarrow Q_a$)

$= \Phi(Q_a)(s_c)$ (by the definition of Φ) .

Next, we define the notion of an *annotated transition.*

Definition A.1. *Let* S *be a set of states and let* $E \subset S \times S$ *be a set of transitions on* S. *An annotated transition is a transition* $e \in E$ *accompanied by a one-state predicate* Pre_e *on* S *and a two-state predicate* $Post_e$ *on* S.

Now, we can state the theorem formally:

Theorem A.2. Let **A,B** and **C** be automata with state spaces S_b, S_b, and S_c and sets of annotated transitions E_a, E_b, and E_c, respenctively. Let α_2: $S_c \rightarrow S_b$ and $E_c \rightarrow E_b$ and α_1: $S_b \rightarrow S_a$ and $E_b \rightarrow E_a$ be refinement mappings,that is, mappings that, together with their induced mappings Φ_2 and Φ_1 on predicates and the transition annotations, satisfy the appropriate analogs of conditions 1,2 and 3. For convenience, we refer to those conditions as conditions $1_{b,c}$, $2_{b,c}$ and $3_{b,c}$ and conditions $1_{a,b}$, $2_{a,b}$ and $3_{a,b}$. Then if $\alpha \triangleq \alpha_1 \circ \alpha_2$ and $\Phi \triangleq \Phi_2 \circ \Phi_1$, the mappings α and Φ satisfy conditions 1,2 and 3, and hence, α: $S_c \rightarrow S_a$, and $E_b \rightarrow E_a$ is a refinement mapping.

Proof. Suppose that the hypotheses of Theorem A.2 hold. Then, we need to establish that conditions 1,2, and 3 hold. For condition 1, we can argue as follows:

(i) Pre_{e_c} => Φ_2 (Pre_{e_b}) (by condition $1_{b,c}$)

(ii) Pre_{e_b} => $\Phi_1(Pre_{e_a})$)(by condition $1_{a,b}$)

(iii) $\Phi_2(Pre_{e_b})$ => $\Phi_2(\Phi_1(Pre_{e_a}))$ (by (ii) and Lemma A.1) and, therefore,

(iv) Pre_{e_c}) => $\Phi_2(\Phi_1(Pre_{e_a}))$ (by (i) and (iii))

(v) Pre_{e_c} => $\Phi(Pre_{e_a})$ (by the definition of Φ)

For condition 2, first note that the first part of condition 2, which relates Pre_{e_c} to $Post_{e_c}$, follows from the first part of condition $2_{b,c}$. The remainder of the argument, which relates $Post_{e_c}$ to $Post_{e_a}$, is totally analogous to that for condition 1.

To prove condition 3, we note that if Pre_{e_c} (s_c) holds, then by condition $3_{b,c}$, the lower square in Fig. 3 commutes. Furthermore, we have

$Pre_{ec}(s_c)$

=> Φ_2 (Pre_{e_b})(s_c) (by condition $1_{b,c}$)

≡ Pre_{e_b} (s_b) (by definition of Φ_2, since $s_c = a_2(s_b)$)

And hence Pre_{e_b} (s_b) holds. By condition $3_{a,b}$, this implies that the upper square commutes. Therefore, the diagram as a whole commutes and we have

$$e_a \circ \alpha_1 \circ \alpha_2 = \alpha_1 \circ \alpha_2 \circ e_c$$

By the definition of a, this means that

$$e_a \circ \alpha = \alpha \circ e_c$$

And we are done.

TAME Representation of Separation

To provide some details of the TAME representation of ED, we show how three of the five subproperties of the separation property verified for ED, Temporal Separation, No-Exfiltration, and No-Infiltration, are represented in TAME. For each subproperty, we first repeat its natural language representation and then show and explain its TAME representation.

B.1 Temporal Separation

Natural language version

(Temporal Separation) For all states s in *S,* for all i, $1 \leq i \leq n$, if the partition id c_s is 0, then the k data areas of partition i are clear, that is, $D_{i,s}^1 = 0, \ldots, D_{i,s}^k = 0$.

TAME version

```
Inv_ClearPart(s:states):bool =
   (FORALL (i:PartIndex): (NONE? (PartId(s)) =>
      (FORALL (n:DataAreaIndex):
         Clear? (MemContent(DataArea(i,n),s)))));

lemma_ClearPart: LEMMA (FORALL (s:states):
    reachable(s) => Inv_ClearPart(s));
```

The TAME representation of the Temporal Separation property is the state-invariant lemma **lemma_ClearPart**, which states that the invariant **Inv_ClearPart** holds for every reachable state **s**. In the invariant **Inv_ClearPart**, **PartIndex** and **DataAreaIndex** are the types of partition indices and data area indices, defined simply to be nonempty, uninterpreted types. Thus, there can be an arbitrary nonzero number of

partitions, each with the same but arbitrary nonzero number of data areas. **PartId(s)** represents c_s, the current partition id in the current state. **NONE?(PartId(s))** is true when c_s is 0, that is, exactly when no partition processing is taking place. **MemContent** is a function that maps a memory area and a state to the memory content of that memory area in that state. Finally, the predicate **Clear?** is true of the memory content of a data area when that data area is clear.

B.2 No-Exfiltration

Natural language version

(No-Exfiltration) Suppose that states s and s' are in state set S, event e is in H, memory area a is in M, and j is a partition, $1 \leq j \leq n$. Suppose further that s'=T(e, s). If e is an event in $P_j \cup E_j^{In} \cup E_j^{Out}$ *and* $a_s \neq a_{s'}$, *then a is in* A_j

TAME version

```
No_Exfiltration: LEMMA
   (FORALL (E:actions, s:states, m:MemAreas, j:PartIndex):
      (enabled (E,s) & Isin(m,PartMemAreas(j)) &
        (NONE? (PartId(s)) OR
          (Part? (PartId(s)) & NOT(Id(PartId(s))=j))))
     => ((InBuff? (E) & InBuff_Index(E) =j) OR
        (OutBuff? (E) & OutBuff_Index(E)=j) OR
       MemContent(m,s)=MemContent (m,trans(E,s))));
```

The TAME version **No_Exfiltration** of the No-Exfiltration property corresponds to the contrapositive of the natural language version. In the TAME representation, the event e is represented by an action E. The state s is represented by **s** and the state s' is represented by **trans(E , s)**, that is, the result of a transition due to action **E** in state **s**. For the current partition id **PartId(s)** in state **s**, either **NONE?** holds, that is, no partition processing is occurring, or **Part?** holds, in which case, partition processing is occurring in partition **id(PartId(s))**. The assertion **enabled(E , s)** means that the precondition of action **E** holds in state **s**. When **E** is an internal action, this precondition ensures that **E** is an internal action for Partition **PartId(s)**. The condition **InBuff?(E)&InBuff _Index(E) - j** is true when **E** fills the input buffer of Partition **j** . The analogous condition with **Out** in place of **In** is true when

E empties the output buffer of Partition **j** . These parts of the conclusion of property **No_Exfiltration** cover the cases when action **E** is an external event for Partition **j** . Thus, property No_Exfiltration says that, if m is a memory area in Partition j and E either is an external action or is an internal action in some partition other than Partition j , then either E is an external action for Partition j or E does not change the content of m.

B.3 No-Infiltration

Natural language version

(No-Infiltration) Suppose that states s_1, s_2, s_1', and s_2' are in S, event e is in H, and i is a partition, $1 \leq i \leq n$. Suppose further that $s_1' = T(e, s_1)$ and $s_2' = T(e, s_2)$. If, for all a in A_i, $a_{s_1} = a_{s_2}$, then, for all a in A_i, $a_{s_1'} = a_{s_2'}$.

TAME version

```
No  Infiltration: LEMMA
 ( FORALL (E:actions, s1, s2:states, m:MemAreas, i:PartIndex):
          enabled(E,s1) & enabled(E,s2) &
          Part? (PartId(s1)) & Id(PartId(s1))=i &
          Part? (PartId(s2)) & Id(PartId(s2))=i &
          Isin(m, PartMemAreas (i)) &
          (FORALL (m1:MemAreas):Isin(m1,PartMemAreas(i))
             => MemContent (m1,s1)=MemContent (m1, s2))
    =>MemContent(m,trans(E,s1))=MemContent(m,trans(E,s2)));
```

The preceding explanation of the notation in lemma_ClearPart and No_Exfiltration should make it clear that the TAME version No Infiltration of the No-Infiltration Property is equivalent to the natural language version.

Tasks for laboratory work №2.

1. Each student choose different type of software.
2. Make the procedure of the code annotation with preconditions and postconditions.
3. Partition the code into the concepts of Event, Trusted, and Other Code. Finally.
4. Demonstrate the conformance of the Event Code and the code preconditions and postconditions with the internal events, preconditions, and postconditions of the TLS.

5. Prove, that the Trusted Code and the Other Code are benign.
6. use model checkers and theorem provers for verifying that a formal specification satisfies a security property of interest.
7. Automatically generate test cases that check source code annotations; automatically construct efficient provably correct code from specifications.

Requirements to the report

The report should consists of:

- title sheet;
- the aim and the task of the laboratory work;
- partitioned code into the concepts of Event, Trusted, and Other Code. Finally;
- demonstration of the conformance of the Event Code and the code preconditions and postconditions with the internal events, preconditions, and postconditions of the TLS;
- Prove, that the Trusted Code and the Other Code are benign;
- results of the verifying that a formal specification satisfies the security property of interest by the usage of the model checkers;
- generated test cases and constructed efficient provably correct code from specifications;
- conclusions.

Advancement questions

1. How to build a well-defined security property?
2. What we should do to build the minimal state machine model?
3. How we can prove that the security model satisfies the property using a mechanical verifier?
4. What should we do to annotate the code with preconditions and postconditions and partition it into Event, Trusted, and Other Code?
5. How to demonstrate conformance of the Event Code and the code preconditions and postconditions with the internal events and preconditions and postconditions of the TLS?
6. What we should do to show that the Trusted Code and the Other Code are benign?

7. How to develop tools for validating and constructing preconditions and postconditions from the source code, including the C code?
8. What we should do to develop tools for automatically generating test cases that check C code annotations?
9. How to develop tools for showing conformance of annotated code with a TLS, and automatically constructing efficient provably correct code from specifications?
10. What does number of international organizations establish to provide a single basis for evaluating the security of information technology products?
11. What are the five steps of the code verification process?
12. What are the main goals of the Top-Level Specification?

2 FORMAL METHODS FOR THE ANALYSIS OF SECURITY PROTOCOLS

2.1 Laboratory work №3. Using Horn Clauses for Analyzing Security Protocols

Theaim and the task of the laboratory work

The aim of this laboratory work is to get acquainted with a method for verifying security protocols based on an abstract representation of protocols by Horn clauses.

Task of the work:

- use the protocol verifier ProVerif.
- define cryptographic primitives defined via rewrite rules or equations.
- prove security properties, including authentication and process equivalences.
- prove security properties of protocols for an unboundednumber of sessions, in a fully automatic way.

Preparation for laboratory work

- to clarify the aims and objectives;
- to study theoretical material given in the description.

Theoretical material

Introduction

Security protocols can be verified by an approach based on Horn clauses; the main goal of this approach is to prove security properties of protocols in the Dolev-Yao model in a fully automatic way without bounding the number of sessions or the message space of the protocol [27]. In contrast to the case of a bounded number of sessions in which decidability results could be obtained, the case of an unbounded number of sessions is undecidable for a reasonable model of protocols [28]. Possible solutions to this problem are relying on user interaction, allowing non-termination, and performing sound approximations (in which case the technique is incomplete: correct security properties cannot always be proved). Theorem proving [29] and rely on user interaction or on manual proofs. Typing generally relies on lightweight user annotations and is incomplete. Strand spaces and rank functions

also provide techniques that can handle an unbounded number of sessions at the cost of incompleteness.

Many methods rely on sound abstractions: they overestimate the possibilities of attacks, most of the time by computing an overapproximation of the attacker knowledge.

They make it possible to obtain fully automatic, but incomplete, systems. The Horn clause approach is one such method. It was first introduced by Weidenbach [30]. Let us consider a variant of this method and extensions that are at the basis of the automatic protocol verifier ProVerif.

In this method, messages are represented by terms; the fact attacker() means that the attacker may have the message; Horn clauses (i.e. logic programming rules) give implications between these facts.

An efficient resolution algorithm determines whether a fact is derivable from the clauses, which can be used for proving security properties. In particular, when attacker() is not derivable from the clauses, the attacker cannot have , that is, is secret. This method is incomplete since it ignores the number of repetitions of each action in the protocol. (Horn clauses can be applied any number of times.) This abstraction is key to avoid bounding the number of runs of the protocol. It is sound, in the sense that if the verifier does not find a flaw in the protocol, then there is no flaw. The verifier therefore provides real security guarantees. In contrast, it may give a false attack against the proto-col. However, false attacks are rare in practice, as experiments demonstrate. Termination is not guaranteed in general, but it is guaranteed on certain subclasses of protocols and can be obtained in all cases by an additional approximation.

Without this additional approximation, even if it does not always terminate and is in-complete, this method provides a good balance in practice: it terminates in the vast majority of cases and is very efficient and precise. It can handle a wide variety of cryptographic primitives defined by rewrite rules or by equations, including shared-key and public-key cryptography (encryption and signatures), hash functions, and the Diffie-Hellman key agreement. It can prove various security properties (secrecy, authentication, and process equivalences).

Other methods rely on abstractions:

- Bolignano [31] was a precursor of abstraction methods for security protocols. He merges keys, nonces, so that only a finite set remains and applies a decision procedure.

- Monniaux [32] introduced a verification method based on an abstract representation of the attacker knowledge by tree automata. This method was extended by Goubault-Larrecq [33]. Genet and Klay [34] combine tree automata with rewriting. This method has lead to the implementation of the TA4SP verifier (*Tree-Automata-based Automatic Approximations for the Analysis of Security Protocols*) [35].

- The main drawback of this approach is that, in contrast to Horn clauses, tree au-tomata cannot represent relational information on messages: when a variable ap-pears several times in a message, one forgets that it has the same value at all its occurrences, which limits the precision of the analysis. The Horn clause method can be understood as a generalization of the tree automata technique. (Tree automata can be encoded into Horn clauses.)

- Control-flow analysis [36,37] computes the possible messages at each program point. It is also non-relational, and merges nonces created at the same program point in different sessions. These approximations make it possible to obtain a complexity at most cubic in the size of the protocol. It was first defined for secrecy for shared-key protocols, then extended to message authenticity and public-key protocols [38], with a polynomial complexity.

- Most protocol verifiers compute the knowledge of the attacker. In contrast, Her-mès [39] computes the form of messages, for instance encryption under certain keys, that guarantee the preservation of secrecy. It handles shared-key and public-key encryption, but the method also applies to signatures and hash functions.

- Backes et al. [40] prove secrecy and authentication by an abstract-interpretation-based analysis. This analysis builds a causal graph that captures the causality between events in the protocol. The security properties are proved by traversing this graph. This analysis always terminates but is incomplete. It assumes that messages are typed, so that names (which represent random numbers) can be distinguished from other messages.

M,N ::=	Terms
x	Variable
a[$M_1, \ldots, M_n$]	name
f ($M_1, \ldots M_n$)	function application
F ::= p($M_1, \ldots, M_n$)	fact
R ::= $F_1 \wedge \ldots \wedge F_n \Rightarrow F$	Horn clause

Figure 3.1. Syntax of protocol representation

One of the first verification methods for security protocols, the Interrogator [41] is also related to the Horn clause approach: in this system, written in Prolog, the reachability of the state after a sequence of messages is represented by a predicate, and the program uses a backward search in order to determine whether a state is reachable or not. The main problem of this approach is non-termination, and it is partly solved by relying on user interaction to guide the search. In contrast, we provide a fully automatic approach by using a different resolution strategy that provides termination in most cases.

The NRL protocol analyzer [42, 43] improves the technique of the Interrogator by using narrowing on rewriting systems. It does not make abstractions, so it is correct and complete but may not terminate.

Abstract Representation of Protocols by Horn Clauses

A protocol is represented by a set of Horn clauses; the syntax of these clauses is given in Figure 1. In this figure, ranges over variables, over names, over function symbols, and over predicate symbols. The terms represent messages that are exchanged between participants of the protocol. A variable can represent any term. Names represent atomic values, such as keys and nonces (random numbers). Each principal has the ability of creating new names: fresh names are created at each run of the protocol. Here, the created names are considered as functions of the messages previously received by the principal that creates the name. Thus, names are distinguished only when the preceding messages are different. As noticed by Martín Abadi (personal communication), this approximation is in fact similar to the approximation done in some type systems (such as [44]): the type of the new name depends on the types in the environment. It is enough to

handle many protocols, and can be enriched by adding other parameters to the name. In particular, by adding as parameter a session identifier that takes a different value in each run of the protocol, one can distinguish all names. This is necessary for proving authentication but not for secrecy, so we omit session identifiers here for simplicity. We refer the reader to [45, 46] for additional information. The function applications $f(M_1, \ldots, M_n)$ build terms: examples of functions are encryption and hash functions. A fact $F = p(M_1, \ldots, M_n)$ expresses a property of the messages $M_1, \ldots, M_n$. Several predicates can be used but, for a first example, we are going to use a single predicate attacker, such that the fact attacker( ) means "the attacker may have the message M". A clause R = $F_1 \wedge \ldots \wedge F_n \Rightarrow F$ means that, if all facts $F_1, \ldots, F_n$, are true, then F is also true. A clause with no hypothesis $\Rightarrow F$ is written simply F.

We use as a running example the naive handshake protocol:

Message 1 $A \rightarrow B : \{|[k]_{sk_A}|\}^a_{pk_B}$

Message 1 $B \rightarrow A : \{|s|\}^s_k$

We denote by sk_A the secret key of A, pk_A his public key, sk_A the secret key of B, pk_B his public key.

Representation of Primitives

Cryptographic primitives are represented by functions. For instance, we represent the public-key encryption by a function pencrypt(m , pk), which takes two arguments: the message to encrypt and the public key pk. There is a function pk that builds the public key from the secret key. (We could also have two functions pk and sk to build respectively the public and secret keys from a secret.) The secret key is represented by a name that has no arguments (that is, there exists only one copy of this name) sk_A [ ] for A and sk_B [ ] for B. Then pk_A = pk(sk_A [ ]) and pk_B = pk(sk_B [ ]).

More generally, we consider two kinds of functions: constructors and destructors. The constructors are the functions that explicitly appear in the terms that represent messages. For instance, pencrypt and pk are constructors. Destructors manipulate terms. A destructor g is defined by a set def(g) of rewrite rules of the form g($M_1, \ldots, M_n$) $\rightarrow$ where $M_1, \ldots, M_n$, are terms that contain only variables and constructors and the variables of M all occur in $M_1, \ldots, M_n$. For

instance, the decryption pdecrypt is a destructor, defined by pdecrypt(pencrypt(m , pk(sk)), sk) → m. This rewrite rule mod-els that, by decrypting a ciphertext with the corresponding secret key, one obtains the cleartext. Other functions are defined similarly:

- For signatures, we use a constructor sign and write sign(m, sk) for the message m signed under the secret key sk. A destructor getmess defined by getmess(sign(m, sk)) → m returns the message without its signature, and checksign(sign(m, sk), pk(sk)) → m returns the message only if the signature is valid.

- The shared-key encryption is a constructor sencrypt and the decryption is a destructor sdecrypt, defined by sdecrypt(sencrypt(m, k), k) → m.

- A one-way hash function is represented by a constructor h (and no destructor).

- Tuples of arity n are represented by a constructor (_, . . . , _) and n destructors $i\text{th}_n$ defined by $i\text{th}_n$ ((x_1, . . . , x_n)) →x_i, i ∈ {1, . . . , n }. Tuples can be used to represent various data structures in protocols.

Rewrite rules offer a flexible method for defining many cryptographic primitives. It can be further extended by using equations.

Representation of the Abilities of the Attacker

We assume that the protocol is executed in the presence of an attacker that can intercept all messages, compute new messages from the messages it has received, and send any message it can build, following the so-called Dolev-Yao model [47]. We first present the encoding of the computation abilities of the attacker.

During its computations, the attacker can apply all constructors and destructors. If f is a constructor of arity n, this leads to the clause:

attacker(x_1) ∧ . . . ∧ attacker(x_n) ⇒ attacker(f (x_1, . . . , x_n)).

If g is a destructor, for each rewrite rule g(M_1, . . . , M_n) → M in def(g), we have the clause:

attacker(M_1) ∧ . . . ∧ attacker(M_n) ⇒ attacker(M).

The destructors never appear in the clauses, they are coded by pattern-matching on their parameters (here M_1, . . . , M_n) in the hypothesis of the clause and generating their result in the conclusion. In the particular case of public-key encryption, this yields:

attacker(m) ∧ attacker(pk) ⇒ attacker(pencrypt(m, pk)),

attacker(sk) ⇒ attacker(pk(sk)),

attacker(pencrypt(m,pk(sk)))∧attacker(sk)⇒attacker(m), (1)

where the first two clauses correspond to the constructors pencrypt and pk, and the last clause corresponds to the destructor pdecrypt. When the attacker has an encrypted message pencrypt(m, pk) and the decryption key sk, then it also has the cleartext m. (We assume that the cryptography is perfect, hence the attacker can obtain the cleartext from the encrypted message only if it has the key.)

Clauses for signatures (sign, getmess, checksign) and for shared-key encryption (sencrypt, sdecrypt) are given in Figure 3.2.

The clauses above describe the computation abilities of the attacker. Moreover, the attacker initially has the public keys of the protocol participants. Therefore, we add the clauses attacker(pk(sk_A [ ])) and attacker(pk(sk_B [ ])). We also give a name to the attacker, that will represent all names it can generate: attacker(a[ ]). In particular, a[ ] can represent the secret key of any dishonest participant, his public key being pk(a[ ]), which the attacker can compute by the clause for constructor pk.

Representation of the Protocol Itself

Now, we describe how the protocol itself is represented. We consider that and are willing to talk to any principal, A, B but also malicious principals that are represented by the attacker. Therefore, the first message sent by A can be pencrypt(sign(k , sk_A [ ]), pk(x)) for any x. We leave to the attacker the task of start-ing the protocol with the principal it wants, that is, the attacker will send a preliminary message to , mentioning the public key of the principal with which should talk. This principal can be , or another principal represented by the attacker. Hence, if the attacker has some key pk(x), it can send pk(x) to A; A replies with his first message, which the attacker can intercept, so the attacker obtains pencrypt(sign(k, (sk_A [ ]), pk(x)). Therefore, we have a clause of the form

attacker(pk(x)) ⇒ attacker(pencrypt(sign(k, (sk_A [ ]), pk(x))).

Moreover, a new key k is created each time the protocol is run. Hence, if two different keys pk(x) are received by A, the generated keys k are certainly different: k depends on pk(x). The clause becomes:

attacker(pk(x)) ⇒ attacker(pencrypt(sign(k[pk(x)], sk_A[ ]), pk(x))). (2)

When B receives a message, he decrypts it with his secret key sk_B, so B pects a message of the form pencrypt (x', pk(sk_B [ ]). Next, B tests whether A signed x', that is, B evaluates checksign(x', pk sk_A), and this succeeds only when x' sign(y, sk_A [ ]). If so, he assumes that the key y is only known by A, and sends a secret s (a constant that the attacker does not have a priori) encrypted under y. We assume t that the attacker relays the message coming from A, and intercepts the message sent by . Hence the clause:

attacker(pencrypt(sign(y,sk_A[]), pk(sk_B [ ]))) ⇒ attacker(sencrypt(s,y)).

Remark 3.1 With these clauses, cannot play the role of B and vice-versa. In order to model a situation in which all principals play both roles, we can replace all occurrences of sk_B [ ] with sk_A[ ] in the clauses above. Then A plays both roles, and is the only honest principal.

More generally, a protocol that contains n messages is encoded by n sets of clauses. If a principal X sends the ith message, the ith set of clauses contains clauses that have as hypotheses the patterns of the messages previously received by X in the protocol, and as conclusion the pattern of the ith message. There may be several possible patterns for the previous messages as well as for the sent message, in particular when the principal X uses a function defined by several rewrite rules, such as the function exp. In this case, a clause must be generated for each combination of possible patterns. More-over, notice that the hypotheses of the clauses describe all messages previously received, not only the last one. This is important since in some protocols the fifth message for instance can contain elements received in the first message. The hypotheses summarize the history of the exchanged messages.

Computation abilities of the attacker:
For each constructor f of arity n:
attacker(x_1)∧ . . . ∧attacker(x_n)⇒attacker(f(x_1, . . . , x_n))
For each destructor g, for each rewrite rule g(M_1, . . . ,M_n) → M in def(g):
that is attacker(M_1) ∧ . . . ∧ attacker(M_n) ⇒ attacker(M)
pencrypt attacker(m) ∧ attacker(pk) ⇒ attacker(pencrypt(m, pk))
pk attacker(sk) ⇒ attacker(pk(sk))
pdecrypt attacker(pencrypt(m, pk(sk))) ∧ attacker(sk) ⇒attacker(m)
sign attacker(m) ∧ attacker(sk) ⇒ attacker(sign(m, sk))
getmess attacker(sign(m, sk)) ⇒ attacker(m)
checksign attacker(sign(m, sk)) ∧ attacker(pk(sk)) ⇒ attacker(m)
sencrypt attacker(m) ∧ attacker(k) ⇒ attacker(sencrypt(m, k))
sdecrypt attacker(sencrypt(m, k)) ∧ attacker(k) ⇒ attacker(m)
Name generation:
attacker(a[ ])
Initial knowledge: attacker(pk(sk_A [ ])), attacker(pk(sk_B[ ]))
The protocol:
First message: attacker(pk(x))
Second message: ⇒
attacker(pencrypt(sign(k[pk(x)], sk_A[ ]), pk(x)))
attacker(pencrypt(sign(y, sk_A[ ]), pk(sk_B[ ])))
⇒ attacker(sencrypt(s,y))

Figure 3.2.Representation of the protocol

Remark 3.2 When the protocol makes some communications on private channels, on which the attacker cannot a priori listen or send messages, a second predicate can be used: message(C,M) meaning "the message M can appear on channel C". In this case, if the attacker manages to get the name of the channel C, it will be able to listen and send messages on this channel. Thus, two new clauses have to be added to describe the behavior of the attacker. The attacker can listen on all channels it has: message(x,y) ∧attacker(x) ⇒ attacker(y). It can send all messages it has on all channels it has: attacker(x) ∧ attacker(y) ⇒ message(x,y).

Summary

To sum up, a protocol can be represented by three sets of Horn clauses, as detailed in Figure 3.2 for the protocol:

- Clauses representing the computation abilities of the attacker: constructors, destructors, and name generation.

- Facts corresponding to the initial knowledge of the attacker. In general, there are facts giving the public keys of the participants and/or their names to the attacker.

- Clauses representing the messages of the protocol itself. There is one set of clauses for each message in the protocol. In the set corresponding to the ith message, sent by principal , the clauses are of the form attacker(M_{j1}) ∧ . . . ∧ attacker(M_{jn}) ⇒ attacker(M_i) where $M_{j1}, \ldots M_{jn}$, are the patterns of the messages received by X before sending the ith message, and M_i is the pattern of the ith message.

Approximations

Specifically, the number of repetitions of each action is ignored, since Horn clauses can be applied any number of times. So a step of the protocol can be completed several times, as long as the previous steps have been completed at least once between the same principals (even when future steps have already been completed). For instance, consider the following protocol (communicated by Véronique Cortier)

First step: A sends $\{|\langle N_1, M\rangle|\}_k^s$ $\{|\langle N_2, M\rangle|\}_k^s$

Second step:

If receives $\{|\langle x, M\rangle|\}_k^s$, he replies with x

Third step: If receives N_1, N_2 he replies with s

where N_1, N_2, and are nonces. In an exact model, never sends s, since $\{|\langle N_1, M\rangle|\}_k^s$ or $\{|\langle N_2, M\rangle|\}_k^s$ can be decrypted, but not both. In the Horn clause model, even though the first step is executed once, the second step may be executed twice for the same (that is, the corresponding clause can be applied twice), so that both $\{|\langle N_1, M\rangle|\}_k^s$ and $\{|\langle N_2, M\rangle|\}_k^s$ can be decrypted, and may send s. We have a false attack against the secrecy of s.

However, the important point is that the approximationsare sound: if an attack exists in a more precise model, such as the applied pi calculus [48] or multiset rewriting [49]. This is shown for the applied pi

calculus in [50] and for multiset rewriting in [30]. In particular, it has shown formally that the only approximation with respect to the multiset rewriting model is that the number of repetitions of actions is ignored. Performing approximations enables us to build a much more efficient verifier, which will be able to handle larger and more complex protocols. Another advantage is that the verifier does not have to limit the number of runs of the protocol. The price to pay is that false attacks may be found by the verifier: sequences of clause applications that do not correspond to a protocol run, as illustrated above. False attacks appear in particular for protocols with temporary secrets: when some value first needs to be kept secret and is revealed later in the protocol, the Horn clause model considers that this value can be reused in the beginning of the protocol, thus breaking the protocol. When a false attack is found, we cannot know whether the protocol is secure or not: a real attack may also exist. A more precise analysis is required in this case. Fortunately, the representation is precise enough so that false attacks are rare.

Secrecy Criterion

Our goal is to determine secrecy properties: for instance, can the attacker get the secret s? That is, can the fact attacker(s) be derived from the clauses? If attacker(s) can be derived, the sequence of clauses applied to derive attacker(s) will lead to the description of an attack.

The notion of secrecy is that a term M is secret if the attacker cannot get it by listening and sending messages, and performing computations. This notion of secrecy is weaker than non-interference, but it is adequate to deal with the secrecy of fresh names. Non-interferenceis better at excluding implicit information flows or flows of parts of compound values.

In example, attacker(s) is derivable from the clauses. The derivation is as follows. The attacker generates a fresh name a[ ] (considered as a secret key), it computes pk(a[ ]) by the clause for pk, obtains pencrypt(sign(k[pk(a[])], sk_A[ ]), pk(a[ ])) by the clause for the first message. It decrypts this message using the clause for pdecrypt and its knowledge of a[ ], thus obtaining sign(k[pk(a[ ])], sk_A[ ]). It reencrypts the sig-nature under pk(sk_B [ ]) by the clause for pencrypt (using its initial knowledge of pk(sk_B [ ])), thus obtaining pencrypt(sign(k[pk(a[ ])], sk_A[ ]), pk(sk_B[ ])). By the clause for the second message, it obtains

sencrypt(s, k[pk(a[ ])]). On the other hand, from sign(k[pk(a[ ])], sk_A[ ]), it obtains k[pk(a[ ])] by the clause for getmess, so it can de-crypt sencrypt(s, k[pk(a[ ])]) by the clause for sdecrypt, thus obtaining s. In other words, the attacker starts a session between A and a dishonest participant of secret key a[]. It gets the first message pencrypt(sign(k, sk_A [ ]), pk(a[ ])), decrypts it, reencrypts it under pk(sk_B [ ]), and sends it to B. For B, this message looks like the first message of a ses-sion between A and B, so B replies with sencrypt(s,k), which the attacker can decrypt since it obtains from the first message. Hence, the obtained derivation corresponds to the known attack against this protocol. In contrast, if we fix the protocol by adding the public key of B in the first message $\{|[\langle pk_B, k\rangle]_{sk_A}|\}^{a}_{pk_B}$, attacker(s) is not derivable from the clauses, so the fixed protocol preserves the secrecy of s.

Next, we formally define when a given fact can be derived from a given set of clauses. Technically, the hypotheses $F_1, \ldots, F_n$ of a clause are considered as a multiset. This means that the order of the hypotheses is irrelevant, but the number of times a hypothesis is repeated is important. (This is not related to multiset rewriting models of protocols: the semantics of a clause does not depend on the number of repetitions of its hypotheses, but considering multisets is necessary in the proof of the resolution algorithm.) We use R for clauses (logic programming *rules*),H for hypothesis, and C for conclusion.

Definition 3.1 (Subsumption) We say that $\boldsymbol{H_1 \Rightarrow C_1}$ subsumes $\boldsymbol{H_2 \Rightarrow C_2}$, and we write $(\boldsymbol{H_1 \Rightarrow C_1}) \sqsupseteq (\boldsymbol{H_2 \Rightarrow C_2})$, if and only if there exists a substitution such that $\boldsymbol{\sigma C_1 = C_2}$ and $\boldsymbol{H_1 \subseteq H_2}$ (multiset inclusion).

We write $\boldsymbol{R_1 \sqsupseteq R_2}$ when $\boldsymbol{R_2}$ can be obtained by adding hypotheses to a particular instance of $\boldsymbol{R_1}$. In this case, all facts that can be derived by $\boldsymbol{R_2}$ can also be derived by $\boldsymbol{R_2}$.

A derivation is defined as follows, as illustrated in Figure 3.3.

Definition 3.2 (Derivability) Let F be a closed fact, that is, a fact without variable. Let R be a set of clauses. F is derivable from R if and only if there exists a derivation of F from R, that is, a finite tree defined as follows:

- Its nodes (except the root) are labeled by clauses R ∈R;
- Its edges are labeled by closed facts;

- If the tree contains a node labeled by R with one incoming edge labeled by F_0 and n outgoing edges labeled by F_1, . . . ,F_n, then R $\sqsupseteq F_1 \wedge \ldots \wedge F_n \Rightarrow \vee F_0$.

- The root has one outgoing edge, labeled by F. The unique son of the root is named the subroot.

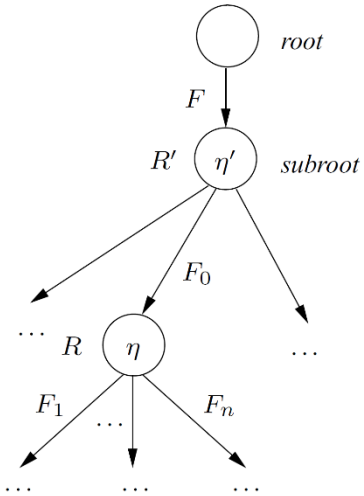

Figure 3.3. Derivation of F

In a derivation, if there is a node labeled by R with one incoming edge labeled by F_0 and n outgoing edges labeled by F_1, . . . , F_n , then F_0 can be derived from F_1, . . . F_n, by the clause R. Therefore, there exists a derivation of F from R if and only if F can be derived from clauses in R (in classical logic).

Resolution Algorithm

The representation is a set of Horn clauses, and our goal is to determine whether a given fact can be derived from these clauses or not. This is exactly the problem solved by usual Prolog systems. However, we cannot use such systems here, because they would not terminate. For instance, the clause:

attacker(pencrypt(m, pk(sk))) ∧ attacker(sk) ⇒ attacker(m)

leads to considering more and more complex terms, with an unbounded number of encryptions. We could of course limit arbitrarily

the depth of terms to solve the problem, but we can do much better than that.

As detailed below, the main idea is to combine pairs of clauses by resolution, and to guide this resolution process by a selection function: our resolution algorithm is resolution with free selection [52]. This algorithm is similar to ordered resolution with selection but without the ordering constraints.

Notice that, since a term is secret when a fact is *not* derivable from the clauses, soundness in terms of security (if the verifier claims that there is no attack, then there is no attack) corresponds to the completeness of the resolution algorithm in terms of logic programming (if the algorithm claims that a fact is not derivable, then it is not). The resolution algorithm that we use must therefore be complete.

The Basic Algorithm

Let us first define resolution: when the conclusion of a clause R unifies with a hypothesis of another (or the same) clause R′, resolution infers a new clause that corresponds to applying R and R′ one after the other. Formally, resolution is defined as follows:

Definition 3.3 Let R and R′ be two clauses, $\mathbf{R} = H \Rightarrow \mathbf{C}$, and $\mathbf{R}' = \mathbf{H}' \Rightarrow \mathbf{C}'$. Assume that there exists $F_0 \in H'$ such that C and F_0 are unifiable and σ is the most general unifier of C and F_0. In this case, we define $R \circ_{f_0} R' = \sigma (H \cup (H' \setminus \{F_0\})) \Rightarrow \sigma C'$. The clause $R \circ_{f_0} R'$ is the result of resolving R' with R upon F_0.

For example, if R is the clause (2), R' is the clause (1), and the fact F_0 is F_0 = attacker(pencrypt(m, pk(sk))), then $R \circ_{f_0} R'$ is

attacker(pk(x)) ∧ attacker(x) ⇒ attacker(sign(k[pk(x)], sk_A [ ]))

with the substitution σ = {sk ↦ x, m ↦ sign(k[pk(x)], sk_A[ ])}.

We guide the resolution by a selection function:

Definition 3.4 A selection function sel is a function from clauses to sets of facts, such that sel(H ⇒ C) ⊆ H. If F ∈ sel(R), we say that F is selected in R. If sel( ) = ∅, we say that no hypothesis is selected in R, or that the conclusion R of is selected.

The resolution algorithm is correct (sound and complete) with any selection function, as we show below. However, the choice of the selection function can change dramatically the behavior of the algorithm. The essential idea of the algorithm is to combine clauses by

resolution only when the facts unified in the resolution are selected. We will therefore choose the selection function to reduce the number of possible unifications between se-lected facts. Having several selected facts slows down the algorithm, because it has more choices of resolutions to perform, therefore we will select at most one fact in each clause. In the case of protocols, facts of the form attacker(x), with x variable, can be unified will all facts of the form attacker(M). Therefore, we should avoid selecting them. So a basic selection function is a function sel_0 that satisfies the constraint

$$sel_0(H \Rightarrow C) = \begin{cases} \emptyset & \text{if } \forall F \in H, \exists x \text{ variable}, F = \text{attacker}(x) \\ \{F_0\} & \text{where } F_0 \in H \text{ and } \forall x \text{ variable}, F_0 \neq \text{attacker}(x) \end{cases}$$

The resolution algorithm works in two phases, described in Figure 3.4. The first phase transforms the initial set of clauses into a new one that derives the same facts. The second phase uses a depth-first search to determine whether a fact can be derived or not from the clauses.

The first phase, saturate(R_0), contains 3 steps.

- The first step inserts in R the initial clauses representing the protocol and the attacker (clauses that are in R_0), after elimination of subsumed clauses by elim: if R' subsumes R, and R and R' are in R, then R is removed by elim(R).

- The second step is a fixpoint iteration that adds clauses created by resolution. The resolution of clauses R and R' is added only if no hypothesis is selected in R and the hypothesis F_0 of R' that we unify is selected. When a clause is created by resolution, it is added to the set of clauses R. Subsumed clauses are eliminated from R.

- At last, the third step returns the set of clauses of R with no selected hypothesis.

Basically, saturate preserves derivability (it is both sound and complete):

First phase: saturation saturate(R_0) =

R ← ∅.

For each R ∈R_0, R ← elim({R} ∪R).

Repeat until a fixpoint is reached

for each R ∈R such that sel(R) = ∅,

for each R' ∈R, for each F_0 ∈ sel(R') such that $R \circ_{f_0} R'$ is defined,

R ← elim($\{R \circ f0\ R'\} \cup$R).
Return {R ∈ R | sel(R) = ∅ }.
Second phase: backward depth-first search

$$\mathsf{deriv}(R, \mathcal{R}, \mathcal{R}_1) = \begin{cases} \emptyset & \text{if } \exists R' \in \mathcal{R}, R' \sqsupseteq R \\ \{R\} & \text{otherwise, if } sel(R) = \emptyset \\ \bigcup\{\mathsf{deriv}(R' \circ_{F_0} R, \{R\} \cup \mathcal{R}, \mathcal{R}_1) \mid R' \in \mathcal{R}_1, \\ \quad F_0 \in sel(R) \text{ such that } R' \circ_{F_0} R \text{ is defined}\} & \text{otherwise} \end{cases}$$

$$\mathsf{derivable}(F, \mathcal{R}_1) = \mathsf{deriv}(F \Rightarrow F, \emptyset, \mathcal{R}_1)$$

Figure 3.4. Resolution algorithm

Lemma 3.1 (Correctness of saturate**)** Let F be a closed fact. F is derivable from R_0 if and only if it is derivable from saturate(R_0).

This result is proved by transforming a derivation of F from R_0 into a derivation of F from saturate(R_0). Basically, when the derivation contains a clause R' with sel(R') ≠ ∅, we replace in this derivation two clauses R, with sel(R) ≠ ∅, and R' that have been combined by resolution during the execution of saturate with a single clause $R \circ f0\ R'$. This replacement decreases the number of clauses in the derivation, so it terminates, and, upon termination, all clauses of the obtained derivation satisfy $sel(R') = \emptyset$ so they are in saturate(R_0).

Usually, resolution with selection is used for proofs by refutation. That is, the nega-tion of the goal is added to the clauses, under the form of a clause without conclusion: F ⇒. The goal F is derivable if and only if the empty clause "⇒" can be derived. Here, we would like to avoid repeating the whole resolution process for each goal, since in general we prove the secrecy of several values for the same protocol. For non-closed goals, we also want to be able to know which instances of the goal can be derived. That is why we prove that the clauses in saturate(R_0) derive the same facts as the clauses in R_0. The set of clauses saturate(R_0) can then be used to query several goals, using the second phase of the algorithm described next.

The second phase searches the facts that can be derived from R_1 = saturate(R_0). This is simply a backward depth-first search. The call derivable(F, R_1) returns a set of clauses R = H ⇒C with no selected hypothesis, such that R can be obtained by resolution from R_1, C is an instance of F, and all instances of F derivable from R_1 can be derived

by using as last clause a clause of derivable(F, R_1). (Formally, if F' is an instance of F derivable from R_1, then there exist a clause H ⇒C∈ derivable(F, R_1) and a substitution σ such that F' = σC and σH is derivable from R_1.)

The search itself is performed by deriv(R, R, R_1). The function deriv starts with R = F ⇒ F and transforms the hypothesis of R by using a clause R' of R_1 to derive an element F_0 of the hypothesis of R. So R is replaced with $R \circ f0\ R'$ (third case of the definition of deriv). The fact F_0 is chosen using the selection function sel. (Hence deriv derives the hypothesis of R using a backward depth-first search. At each step, the clause R can be obtained by resolution from clauses of R_1, and R concludes an instance of F.) The set R is the set of clauses that we have already seen during the search. Initially, R is empty, and the clause R is added to R in the third case of the definition of deriv.

The transformation of R described above is repeated until one of the following two conditions is satisfied:

- R is subsumed by a clause in R: we are in a cycle; we are looking for instances of facts that we have already looked for (first case of the definition of deriv);
- sel(R) is empty: we have obtained a suitable clause R and we return it (second case of the definition of deriv).

Intuitively, the correctness of derivable expresses that if F', instance of F, is derivable, then F' is derivable from R_1 by a derivation in which the clause that concludes F' is in derivable(F, R_1).

Lemma 3.2 (Correctness of derivable**)** Let F' be a closed instance of F. F' is derivable from R_1. if and only if there exist a clause H ⇒C in derivable(F, R_1) and a substitution σ such that $\sigma C = F'$ and all elements of σH are derivable from R_1.

Basically, this result is proved by transforming a derivation of F' from R_1 into a derivation of F' whose last clause (the one that concludes F') is H ⇒C and whose other clauses are still in R_1. The transformation relies on the replacement of clauses combined by resolution during the execution of derivable.

It is important to apply saturate before derivable, so that all clauses in R_1 have no selected hypothesis. Then the conclusion of these clauses is in general not attacker(x) (with the optimizations and a selection

function that satisfies (3), it is never attacker(x)), so that we avoid unifying with attacker(x).

The following theorem gives the correctness of the whole algorithm. It shows that we can use algorithm to determine whether a fact is derivable or not from the initial clauses. The first part simply combines Lemmas 1 and 2. The second part mentions two easy and important particular cases.

Theorem 3.1 (Correctness) Let F' be a closed instance of F. F' is derivable from R_0 if and only if there exist a clause H ⇒C in derivable(F, saturate(R_0)) and a substitution σ such that σ C=F' and all elements of σH are derivable from saturate(R_0).

In particular, if derivable(F, saturate(R_0)) $\neq \emptyset$, then no instance of F is derivable from saturate(R_0). If the selection function satisfies (3) and F is closed, then F is derivable from R_0 if and only if derivable(F, saturate(R_0))$\neq \emptyset$.

Proof:

The first part of the theorem is obvious from Lemmas 3.1 and 3.2. The first particular case is also an obvious consequence. For the second particular case, if F is derivable from R_0, then derivable(F, saturate(R_0)) $\neq \emptyset$ by the first particular case. For the converse, suppose that derivable(F, saturate(R_0)) $\neq \emptyset$. Then derivable(F, saturate(R_0)) contains a clause H ⇒ C. By definition of derivable, C is an instance of F, so C = F , and sel(H ⇒C) $\neq \emptyset$, so all elements of are of the form attacker(x_i) for some variable x_i. The attacker has at least one term M, for instance a[ ], so attacker(σx_i) is derivable from R_0, where σx_i = M. Hence all elements of σ H are derivable from R_0, so from saturate(R_0), and $\sigma C = F$. Therefore, F is derivable from R_0.

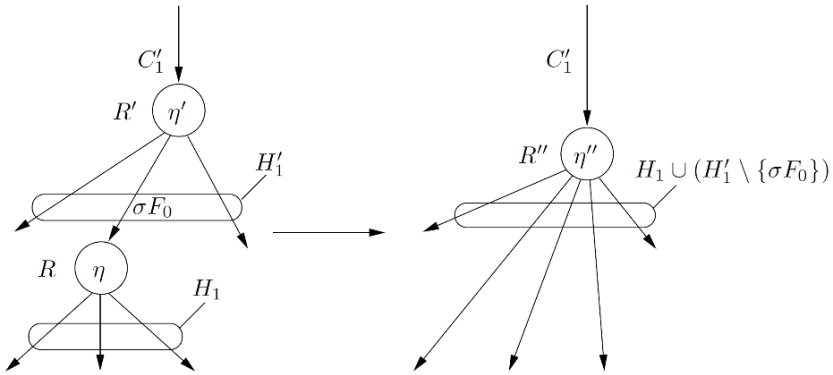

Figure 3.5. Merging of nodes of Lemma 3.3

Proofs

Let us consider the proofs of Lemmas 3.1 and 3.2. We first need to prove a few preliminary lemmas. The first one shows that two nodes in a derivation can be replaced by one when combining their clauses by resolution.

Lemma 3.3 Consider a derivation containing a node η' , labeled R'. Let F_0 be a hypothesis of R' . Then there exists a son η of η' , labeled R, such that the edge η' →η is labeled

by an instance of F_0, $R\ 0_{F_0}$R' is defined, and one obtains a derivation of the same fact by replacing the nodes η and η' with a node η" labeled R"= $R\ 0_{F_0}$R'.

Proof:

This proof is illustrated in Figure 3.5. Let R' = H' ⇒ C', H_1' be the multiset of the labels of the outgoing edges of η', and C_1' the label of its incoming edge. We have R' ⊒ ($H_1' \Rightarrow C_1'$)

so there exists a substitution σ such that $\sigma H' \subseteq H_1'$ and $\sigma C' = C_1'$. Since $F_0 \in H', \sigma F_0 \in H_1'$, so there is an outgoing edge of η' labeled σF_0. Let η be the node at the end of this edge, let R = H ⇒ C be the label of η. We rename the variables of R so that they are distinct from the variables of R' . Let H_1 be the multiset of the labels of the outgoing edges of η. So R⊒ ($H_1 \Rightarrow \sigma F_0$). By the above choice of distinct variables, we can then extend σ so that σH ⊆H_1 and = $C = \sigma F_0$.

The edge η' →η is labeled σF_0, instance of F_0. Since = $\sigma C = \sigma F_0$, the facts C and F_0 are unifiable, so $R \circ_{F_0} R'$ is defined. Let σ' be the most general unifier of C and F_0, and σ'' such that $\sigma = \sigma''\sigma'$. We have $R \circ_{F_0} R' = \sigma'\big(H \cup (H'\backslash\{F_0\})\big) \Rightarrow \sigma' C'$. Moreover, $\sigma''\sigma' C'\big(H \cup (H'\backslash\{F_0\})\big) \subseteq H_1 \cup (H'\backslash\{\sigma F_0\})$ and $\sigma''\sigma' C' = \sigma C' = C'_1$. Hence $R'' = R \circ_{F_0} R' \sqsupseteq (H_1 \cup (H'\backslash\{\sigma F_0\}) \Rightarrow C'_1$. The multiset of labels of outgoing edges of η" is precisely $H_1 \cup (H'\backslash\{\sigma F_0\})$ and the label of its incoming edge is C'_1 , therefore we have obtained a correct derivation by replacing η and η' with η".

Lemma 3.4 If a node η of a derivation D is labeled by R, then one obtains a derivation D' of the same fact as D by relabeling η with a clause R' such that R' ⊒ R.

Proof:

Let H be the multiset of labels of outgoing edges of the considered node η, and C be the label of its incoming edge. We have R ⊒ H ⇒ C. By transitivity of ⊒, R' ⊒ H ⇒ C. So we can relabel η with R'.

Lemma 3.5 At the end of saturate, R satisfies the following properties:

- For all R ∈ R_0, R is subsumed by a clause in R;
- Let R∈R and R' ∈R. Assume that sel(R) = ∅ and there exists F_0∈ sel(R') such that $R \circ_{F_0} R'$ is defined. In this case, $R \circ_{F_0} R'$ is subsumed by a clause in R.

Proof:

To prove the first property, let R ∈R_0. We show that, after the addition of R to R, R is subsumed by a clause in R.

In the first step of saturate, we execute the instruction R ← elim({R}∪R). After execution of this instruction, R is subsumed by a clause in R.

Assume that we execute R ← elim({R"}∪R) for some clause R" and that, before this execution, R is subsumed by a clause R in R, say R'. If R' is removed by this instruction, there exists a clause R'_1 in R that subsumes R', so by transitivity of subsumption, R'_1 subsumes R, hence R is subsumed by the clause $R'_1 \in R$ after this instruction. If R' is not removed by this instruction, then R is subsumed by the clause $R' \in R$ after this instruction.

Hence, at the end of saturate, R is subsumed by a clause in R, which proves the first property.

In order to prove the second property, we just need to notice that the fixpoint is reached at the end of saturate, so R = elim({$R \circ_{F_0} R'$} ∪R). Hence, $R \circ_{F_0} R'$ is eliminated by elim, so it is subsumed by some clause in R.

Proof of Lemma 3.1:

Assume that F is derivable from R_0 and consider a derivation of F from R_0. We show that F is derivable from saturate(R_0).

We consider the value of F the set of clauses R at the end of saturate. For each clause R in R_0, R is subsumed by a clause in R (Lemma 3.5, Property 3.1). So, by Lemma 3.4, we can replace all clauses R in the considered derivation with a clause in R. Therefore, we obtain a derivation D of F from R.

Next, we build a derivation of F from R_1, where R_1 = saturate(R_0). If D contains a node labeled by a clause not in R_1, we can transform D as follows. Let η' be a lowest node of D labeled by a clause not in R_1. So all sons of η' are labeled by elements of R_1. Let R' be the clause labeling η'. Since $R' \notin R_1$, sel(R')≠ ∅. Take $F_0 \in sel(R')$. By Lemma 3.3, there exists a son of η of η' labeled by R, such that $R \circ_{F_0} R'$ is defined, and we can replace η and η' with a node η" labeled by $R \circ_{F_0} R'$. Since all sons of η' are labeled by elements of R_1,R ∈R_1. Hence sel(R) = ∅. So, by Lemma 3.5, Property 2, $R \circ_{F_0} R'$ is subsumed by a clause R" in R. By Lemma 3.4, we can relabel η" with R". The total number of nodes strictly decreases since η and η' are replaced with a single node η".

So we obtain a derivation D' of F from R, such that the total number of nodes strictly decreases. Hence, this replacement process terminates. Upon termination, all clauses are in R_1. So we obtain a derivation of F from R_1, which is the expected result.

For the converse implication, notice that, if a fact is derivable from R_1, then it is derivable from R, and that all clauses added to R do not create new derivable facts: if a fact is derivable by applying the clause $R \circ_{F_0} R'$ then it is also derivable by applying R and R'.

Proof of Lemma 3.2:

Let us prove the direct implication. We show that, if F' is derivable from R_1, then there exist a clause H ⇒C in derivable(F, R_1) and a

substitution σ such that σC=F' and all elements of σH are derivable from R_1.

Let D be the set of derivations D' of F' such that, for some R, the clause R' at the subroot of D' satisfies deriv(R' , R, R_1) ⊆ derivable(F, R_1) and ∀R" ∈R, R" $\not\sqsupseteq$ R' and the other clauses of D' are in R_1.

Let D_0 be a derivation of F' from R_1. Let D_0' be obtained from D_0 by adding a node labeled by R' = F ⇒ F at the subroot of D_0. By definition of derivable, deriv(R, ∅,R_1) ⊆ derivable(F, R_1), and ∀R"∈ ∅,R" $\not\sqsupseteq$ R'. Hence D_0' is a derivation of F' in D, so D is non-empty.

Now, consider a derivation D_1 in D with the smallest number of nodes. The clause R' labeling the subroot η' of D_1 satisfies deriv(R, R, R_1) ⊆ derivable(F, R_1), and ∀R"∈R, R" $\not\sqsupseteq$ R'. In order to obtain a contradiction, we assume that sel(R')≠ ∅. Let $F_0 \in sel(R')$. By Lemma 3.3, there exists a son η of η', labeled by R, such that $R \circ_{F_0} R'$ is defined and we can replace η and η' with a node η" labeled by $R_0 = R \circ_{F_0} R'$, obtaining a derivation D_2 of F' with fewer nodes than D_1. The subroot of D_2 is the node η" labeled by R_0.

By hypothesis on the derivation D_1, R ∈R_1, so deriv(R_0,{R'}∪R, R_1) ⊆ deriv(R', R, R_1) ⊆ derivable(F, R_1) (third case of the definition of deriv(R', R, R_1)).

If ∀R_1 ∈ {R'} ∪R, R_1 $\not\sqsupseteq$ R_0, D_2 is a derivation of F' in D, with fewer nodes than D_1, which is a contradiction.

Otherwise, ∃R_1∈ {R'} ∪R, R_1 ⊒R_1. Therefore, by Lemma 3.4, we can build a derivation D_1 by relabeling η" with R_1 in D_2. There is an older call to deriv, of the form deriv(R_1,R', R_1), such that deriv(R_1,R', R_1) ⊆ derivable(F, R_1). Moreover, R_1 has been added to R' in this call, since R_1 appears in {R'} ∪R. Therefore the third case of the definition of deriv(R_1, R', R_1) has been applied, and not the first case. So ∀R_2 ∈R', R_2 $\not\sqsupseteq$ R_1, so the derivation D_3 is in D and has fewer nodes than D_1, which is a contradiction.

In all cases, we could find a derivation in D that has fewer nodes than D_1. This is a contradiction, so sel(R') = ∅, hence deriv(R, R, R_1) = {R'} (second case of the definition of deriv), so R' ∈derivable(F, R_1). The other clauses of this derivation are in R_1. By definition of a derivation, R' ⊒ H' ⇒ F where H' is the multiset of labels of the outgoing edges of the subroot of the derivation. Taking R' = H ⇒ C,

there exists σ such that $\sigma C = F$ and H ⊆ H', so all elements of σH are derivable from R_1.

The proof of the converse implication is left to the reader. (Basically, if a fact is derivable by applying the clause $R \circ_{F_0} R'$, then it is also derivable by applying R and R'.)

Optimizations

The resolution algorithm uses several optimizations, in order to speed up resolution. The first two are standard, while the last three are specific to protocols.

Elimination of duplicate hypotheses If a clause contains several times the same hypotheses, the duplicate hypotheses are removed, so that at most one occurrence of each hypothesis remains.

Elimination of tautologies If a clause has a conclusion that is already in the hypotheses, this clause is a tautology: it does not derive new facts. Such clauses are removed.

Elimination of hypotheses attacker(x) If a clause H ⇒C contains in its hypotheses attacker(x), where is a variable that does not appear elsewhere in the clause, then the hypothesis attacker(x) is removed. Indeed, the attacker always has at least one message, so attacker(x) is always satisfied for some value of x.

Decomposition of data constructors A data constructor is a constructor f of arity n that comes with associated destructors g_i for i ∈ {1, . . . ,n } defined by $g_i(\mathrm{f}(x_1, \ldots, x_n)) \rightarrow x_i$. Data constructors are typically used for representing data structures. Tuples are examples of data constructors. For each data constructor f, the following clauses are generated:

attacker(x_1) ∧...∧ attacker(x_n) ⇒ attacker(f(x_1, . . . , x_n)) (Rf)

attacker(f(x_1,..., x_n))⇒attacker(x_i) (Rg)

Therefore, attacker(f(p_1, . . . , p_n)) is derivable if and only if ∀i ∈{1, . . . ,n}, attacker(p_i) is derivable. When a fact of the form attacker(f(p_1, . . . , p_n)) is met, it is replaced with attacker(p_1)∧. . .∧attacker(p_n). If this replacement is done in the conclusion of a clause H ⇒ attacker(f(p_1, . . . , p_n)), n clauses are created: H ⇒ attacker(p_i) for each i ∈ {1, . . . ,n }. This replacement is of course done recursively: if p_i itself is a data constructor application, it is replaced again. The clauses (Rf) and (Rg) for data constructors are left unchanged. (When

attacker(x) cannot be selected, the clauses (Rf) and (Rg) for data constructors are in fact not necessary, because they generate only tautologies during resolution. However, when attacker(x) can be selected, which cannot be excluded with certain extensions, these clauses may become necessary for soundness.)

Secrecy assumptions When the user knows that a fact will not be derivable, he can tell it to the verifier. (When this fact is of the form attacker(M), the user tells that M remains secret.) The tool then removes all clauses which have this fact in their hypotheses. At the end of the computation, the tool checks that the fact is indeed underivable from the obtained clauses. If the user has given erroneous information, an error message is displayed. Even in this case, the verifier never wrongly claims that a protocol is secure.

Mentioning such underivable facts prunes the search space, by removing useless clauses. This speeds up the resolution algorithm. In most cases, the secret keys of the principals cannot be known by the attacker. So, examples of underivable facts are attacker(sk_A [ ]), attacker(sk_B [ ]), ...

Termination

In general, the resolution algorithm may not terminate. (The derivability problem is un-decidable.) In practice, however, it terminates in most examples.

In [53] it is shown that it always terminates on a large and interesting class of protocols, the *tagged protocols* [53]. We consider protocols that use as crypto-graphic primitives only public-key encryption and signatures with atomic keys, shared-key encryption, message authentication codes, and hash functions. Basically, a protocol is tagged when each application of a cryptographic primitive is marked with a distinct constant tag. It is easy to transform a protocol into a tagged protocol by adding tags. For instance, example of protocol can be transformed into a tagged protocol, by adding the tags c_0, c_1, c_2 to distinguish the encryptions and signature:

Message 1. $A \rightarrow B \left\{\left|\langle c_1, [\langle c_0, k\rangle]_{sk_A}\rangle\right|\right\}^{a}_{pk_B}$

Message 2. $B \rightarrow A \{|\langle c_2, s\rangle|\}^{s}_{k}$

Adding tags preserves the expected behavior of the protocol, that is, the attack-free ex-ecutions are unchanged. In the presence of attacks,

the tagged protocol may be more secure. Hence, tagging is a feature of good protocol design: the tags are checked when the messages are received; they facilitate the decoding of the received messages and prevent confusions between messages. More formally, tagging prevents type-flaw attacks, which occur when a message is taken for another message. However, the tagged protocol is potentially more secure than its untagged version, so, in other words, a proof of security for the tagged protocol does not imply the security of its untagged version.

Extensions. Treatment of Equations

Up to now, we have defined cryptographic primitives by associating rewrite rules to destructors. Another way of defining primitives is by equational theories, as in the applied pi calculus [54]. This allows us to model, for instance, variants of encryption for which the failure of decryption cannot be detected or more complex primitives such as Diffie-Hellman key agreements. The Diffie-Hellman key agreement enables two principals to build a shared secret. It is used as an elementary step in more complex protocols, such as SSH, SSL, and IPsec.

The Horn clause verification approach can be extended to handle some equational theories. For example, the Diffie-Hellman key agreement can be modeled by using a constant g and a function exp that satisfy the equation

$$\exp(\exp(g,x),y) = \exp(\exp(g,y),x). \quad (4)$$

In practice, the function is exp(x,y) = x^y mod , where is prime and g is a generator of $\mathbb{Z}_p^*$. The equation $\exp(\exp(g,x),y) = (g^x)^y \bmod p = (g^y)^x \bmod p = \exp(\exp(g,y),x)$ is satisfied. In ProVerif, following the ideas used in the applied pi calculus [6], we do not consider the underlying number theory; we work abstractly with the equation (4). The Diffie-Hellman key agreement involves two principals A and B. A chooses a random name x_0, and sends $exp(g,x_0)$ to B. Similarly, B chooses a random name x_1, and sends $exp(g,x_1)$ to A. Then A computes $\exp(\exp(g,x_1),x_0)$ and B computes $exp(exp(g,x_0),x_1)$. Both values are equal by (4), and they are secret: assuming that the attacker cannot have x_0 or x_1, it can compute neither $\exp(\exp(g,x_1),x_0)$ nor $exp(exp(g,x_0),x_1)$.

In ProVerif, the equation (4) is translated into the rewrite rules $exp(exp(g,x),y) \rightarrow exp(exp(g,y),x)$ $exp(x,y) \rightarrow exp(x,y)$.

Notice that this definition of exp is non-deterministic: a term such as $exp(exp(g,a),b)$ can be reduced to $exp(exp(g,b),a)$ and $exp(exp(g,a),b)$, so that $exp(exp(g,a),b)$ re-duces to its two forms modulo the equational theory. The rewrite rules in the definition of function symbols are applied exactly once when the function is applied. So the rewrite rule $exp(x,y) \rightarrow exp(x,y)$ is necessary to make sure that exp never fails, even when the first rewrite rule cannot be applied, and these rewrite rules do not loop because they are applied only once at each application of exp.

This treatment of equations has the advantage that resolution can still use syntactic unification, so it remains efficient. However, it also has limitations; for example, it cannot handle associative functions, such as XOR, because it would generate an infinite number of rewrite rules for the destructors. Recently, other treatments of equations that can han-dle XOR and Diffie-Hellman key agreements with more detailed algebraic relations (in-cluding equations of the multiplicative group modulo p) within the Horn clause approach have been proposed by Küsters and Truderung: they handle XOR provided one of its two arguments is a constant in the clauses that model the protocol [55] and Diffie-Hellman key agreements provided the exponents are constants in the clauses that model the pro-tocol [56]; they proceed by transforming the initial clauses into richer clauses on which the standard resolution algorithm is applied.

Translation from the Applied Pi Calculus

ProVerif does not require the user to manually enter the Horn clauses described previ-ously. These clauses can be generated automatically from a specification of the protocol in the applied pi calculus [6]. On such specifications, ProVerif can verify various security properties, by using an adequate translation into Horn clauses:

- secrecy, as described above. The translation from the applied pi calculus to Horn clauses is given in [50].
- correspondences, which are properties of the form “if an event has been executed, then other events have been executed” [45]. They can in particular be used for formalizing authentication.

- some process equivalences, which mean intuitively that the attacker cannot distinguish two processes (i.e. protocols). Process equivalences can be used for for-malizing various security properties, in particular by expressing that the attacker cannot distinguish a process from its specification. ProVerif can prove particular cases of observational equivalences. It can prove strong secrecy [57], which means that the attacker cannot see when the value of the secret changes. This is a stronger notion of secrecy than the one mentioned previously. It can be used, for instance, for expressing the secrecy of values taken among a set of known constants, such as bits: one shows that the attacker cannot distinguish whether the bit is 0 or 1. More generally, ProVerif can also prove equivalences between processes that differ by the terms they contain, but have otherwise the same structure [58]. In particular, these equivalences can express that a password-based protocol is resistant to guessing attacks: even if the attacker guesses the password, it cannot verify that its guess is correct.

As for secrecy, when no derivation from the clauses is found, the desired security prop-erty is proved. When a derivation is found, there may be attack. ProVerif then tries to reconstruct a trace in the applied pi calculus semantics that corresponds to this derivation [59]. (Trace reconstruction may fail, in particular when the derivation corresponds to a false attack; in this case, one does not know whether there is an attack or not.)

Application to Examples of Protocols

The automatic protocol verifier ProVerif is available at http://www.proverif. ens.fr/. It is successfully applied to many protocols of the literature, to prove secrecy and authentication properties: flawed and corrected versions of the Needham-Schroeder public-key [82,74] and shared-key [43,82, 83], Woo-Lam public-key [88,89] and shared-key [9, 12, 62,88,89], Denning-Sacco [54,9], Yahalom [43], Otway-Rees [9,84, 85], and Skeme [70] protocols. No false attack occurred in the tests and the only non-termination cases were some flawed versions of the Woo-Lam shared-key protocol. The protocols can be verified in less than one second [31].

ProVerif is also used for proving strong secrecy in the corrected version of the Needham-Schroederpublic-keyprotocol [74] and in the Otway-Rees [84], Yahalom [43], and Skeme [70] protocols, the

resistance to guessing attacks for the password-based protocols EKE [18] and Augmented EKE [20], and authentication in the Wide-Mouth-Frog protocol [8] (version with one session). The runtime goes from less than one second to 15 s on the tests [29,34].

Moreover, ProVerif is also used in more substantial case studies:

- With Abadi [4], is applied to the verification of a certified email protocol [7]. We can use correspondence properties to prove that the receiver receives the message if and only if the sender has a receipt for the message. (We can use simple manual arguments to take into account that the reception of sent messages is guaranteed.) One of the tested versions includes the SSH transport layer in order to establish a secure channel.

- With Abadi and Fournet [5], we can study the JFK protocol (*Just Fast Keying*) [10], which was one of the candidates to the replacement of IKE as key exchange proto-col in IPSec. We combined manual proofs and ProVerif to prove correspondences and equivalences.

- With Chaudhuri [35], we can study the secure filesystem Plutus [68] with ProVerif, which allowed us to discover and fix weaknesses of the initial system.

Other authors also use ProVerif for verifying protocols or for building other tools:

- Bhargavan et al. [21,23,27] use it to build the Web services verification tool Tu-laFale: Web services are protocols that send XML messages; TulaFale translates them into the input format of ProVerif and uses ProVerif to prove the desired security properties.

- Bhargavan et al. [24,25,26] use ProVerif for verifying implementations of protocols in F# (a functional language of the Microsoft .NET environment): a sub-set of F# large enough for expressing security protocols is translated into the in-put format of ProVerif. The TLS protocol, in particular, was studied using this technique [22].

- Canetti and Herzog [44] use ProVerif for verifying protocols in the computational model: they show that, for a restricted class of protocols that use only public-key encryption, a proof in the Dolev-Yao model implies security in the computational model, in the universal composability framework. Authentication is verified using correspondences, while secrecy of keys corresponds to strong secrecy.

- ProVerif is also can be used for verifying a certified email web service [75], a certified mailing-list protocol [69], e-voting protocols [16,71], the ad-hoc routing protocol ARAN (*Authenticated Routing for Adhoc Networks*) [61], and zero-knowledge protocols [17].

Finally, Goubault-Larrecq and Parrennes [65] also use the Horn clause method for analyzing implementations of protocols written in C. However, they translate protocols into clauses of the H_1 class and use the H_1 prover by Goubault-Larrecq [64] rather than ProVerif to prove secrecy properties of the protocol.

Conclusion

A strong aspect of the Horn clause approach is that it can prove security properties of protocolsfor an unboundednumber of sessions, in a fully automatic way. This is essential for the certification of protocols. It also supports a wide variety of security primitives and can prove a wide variety of security properties.

On the other hand, the verification problem is undecidable for an unboundednumber of sessions, so the approach is not complete: it does not always terminate and it performs approximations, so there exist secure protocols that it cannot prove, even if it is very precise and efficient in practice.

Tasks for laboratory work №3.

1. According to given in Table 3.1 variant verify security protocols based on an abstract representation of protocols by Horn clauses.
2. Use the protocol verifier ProVerif.
3. Use different sets of sessions for protocol verification.
4. Specify different cryptographic primitives defined by rewrite rules or equations.
5. Prove the security authentication properties .
6. Prove the process equivalences.

Table 3.1. Variants

№	Protocol
1	S/MIME
2	VPN
3	IPSec
4	TLS
5	SSL
6	HTTPS
7	PGP
	S-HTTP
	KERBEROS
	SET

Requirements to the report

The report should consists of:

- title sheet;
- the aim and the task of the laboratory work;
- results of the security protocols verification based on the abstract representation of protocols by Horn clauses;
- results of the usage of the protocol verifier ProVerif;
- results of the usage of the different sets of sessions for protocol verification;
- list of the specified cryptographic primitives defined by rewrite rules or equations;
- presentation of the provement of the the security authentication properties and the process equivalences;
- conclusions.

Advancement questions

1. What we should do to verify security protocols based on an abstract representation of protocols by Horn clauses?
2. How to use the protocol verifier ProVerif?
3. How to use different sets of sessions for protocol verification?
4. What cryptographic primitives can be defined by rewrite rules or equations?

5. How to prove the security authentication properties?
6. How to prove the process equivalences?
7. What is the main goal of Horn clauses?
8. What can be verified by an approach based on Horn clauses and how?
9. What is the the Dolev-Yao model?
10. What do the methods rely on sound abstractions overestimate the possibilities of attacks?

2.2 Laboratory work №4. Validating security protocols under the general attacker

The aim and the task of the laboratory work

The aim of this laboratory work is to analyze the security protocols under the General Attacker threat model.

Task of the work:

- get acquainted with the General Attacker threat model.
- use model checker SATMC to automatically validate a protocol under the new threats, in order to found retaliation and anticipation attacks automatically.

Preparation for laboratory work

- to clarify the aims and objectives;
- to study theoretical material given in the description.

Theoretical material

Introduction

The analysis of security protocols stands among the most attractive niches of research in computer science, as it has attracted efforts from many communities. It is difficult to even provide a satisfactory list of citations, which would have to at least include process calculi, strand spaces, the inductive method and advanced model checking techniques [61-67].

Any meaningful statement about the security of a system requires a clear specification of the threats that the system is supposed to withstand. Such a specification is usually referred to as *threat model* .

Statements that hold under a threat model may no longer hold under other models. For example, if the threat model only accounts for attackers that are outsiders, then Lowe's famous attack on the Needham-Schroeder Public-Key (NSPK) protocol cannot succeed, and the protocol may be claimed secure. But the protocol is notoriously insecure under a model that allows the attacker as a registered principal. The standard threat model for symbolic protocol analysis is the Dolev-Yao model (DY in brief), which sees a powerful attacker control the whole network traffic. The usual justification is that a protocol secure under DY certainly is secure under a less powerful, perhaps more realistic attacker. By contrast, a large group of researchers consider DY insufficient because a DY attacker cannot do cryptanalysis, and their probabilistic reasoning initiated with a foundational research. This sparked off a research thread that has somewhat evolved in parallel with the DY research line, although some efforts exist in the attempt to conjugate them [68]. The present research is not concerned with probabilistic protocol analysis. The main argument is that security protocols may still hide important subtleties even after they are proved correct under DY. These subtleties can be discovered by symbolic protocol analysis under a new threat model that adheres to the present real world more strictly than DY does. The new model we develop here is the *General Attacker* (GA in brief), which features each protocol participant as a DY attacker who does not collude or share knowledge with anyone else. In GA, it is meaningful to continue the analysis of a protocol *after* an attack is mounted, or to anticipate the analysis by looking for extra flaws *before* that attack, something that has never been seen in the relevant literature. This can assess whether additional attacks can be mounted either by the same attacker or by different attackers. Even novel scenarios whereby principals attack each other become possible. A significant scenario is that of *retaliation* [69], where an attack is followed by a counterattack. It recasts that scenario into the new GA threat model. Also, a completely new scenario is defined, that of *anticipation*, where an attack is anticipated, before its termination, by another attack by some other principal.

As its main contribution, the research tailors an existing formalism suited for model checking to accommodate the GA threat model. This makes it possible to analyse protocol subtleties that go beyond standard security properties such as confidentiality and authentication. We begin

by extending an existing setrewriting formalisation of the classical DY model to capture the GA model. Then, we leverage an established model checking tool for security protocols to tackle the validation problems arisen from the new threat model. Finally, we run the tool over the NSPK protocol and its variants to investigate retaliation and anticipation attacks.

The General Attacker

DY can be considered the standard threat model to study security protocols [47]. The DY attacker controls the entire network although he cannot perform cryptanalysis. Some historical context is useful here. The model was defined in the late 1970s when remote computer communications were still confined to military/espionage contexts. It was then natural to imagine that the entire world would collude joining their forces against a security protocol session between two secret agents of the same country. The DY model has remarkably favoured the discovery of significant protocol, but the prototype attacker is significantly changed today. To become an attacker has never been so easy as in the present technological setting because hardware is inexpensive, while security skill is at hand—malicious exploits are even freely downloadable from the web.

A seminal threat model called BUG [69] is recalled here. The name is a permuted acronym for the “Good”, the “Bad” and the “Ugly”. This model attempts stricter adherence than DY’s to the changed reality by partitioning the participants in a security protocol into three groups. The Good principals would follow the protocol, the Bad would in addition try to subvert it, and the Ugly would be ready to either behaviour. This seems the first account in the literature of formal protocol verification on the chance that attackers may attempt to attack each other without sharing knowledge. More recently, Bella observed [70] that the partition of the principals had to be dynamically updated very often, in principle at each event of a principal’s sending or receiving a message, depending on whether the principal respected the protocol or not. Thus, BUG appeared overly detailed, and he simplified it as the Rational Attacker threat model: each principal may at any time make cost/benefit decisions to either behave according to the protocol or not [71]. After BUG’s inception, homologous forms of rational attackers were specifically carved out in the area of game theory and therefore

are, as such, not directly related to the symbolic analysis [71-73]. Analyzing a protocol under the Rational Attacker requires specifying each principal's cost and benefit functions, but this still seems out of reach, especially for classical model checking. By abstracting away the actual cost/benefit analysis, we derive the following simplified model: The General Attacker (GA) threat model: *each principal is a Dolev-Yao attacker.*

The change of perspective in GA with respect to DY is clear: principals do not collude for a common aim but, rather, each of them acts for his own personal sake. Although there is no notion of collusion constraining this model, the human protocol analyser can define some for their particular investigations. The GA model has each principal endowed with the entire potential of a DY attacker. So, each principal may at any stage send any of the messages he can form to anyone. Of course, such messages include both the legal ones, conforming to the protocol in use, and the illegal, forged ones. As we shall see, analysing the protocols under the GA threat model yields unknown scenarios featuring retaliation or anticipation attacks. This paves the way for a future analysis under the Rational Attacker. For example, if an attack can be retaliated under GA, such a scenario will not occur under the Rational Attacker because the cost of attacking clearly overdoes its benefit, and hence the attacker will not attack in the first place.

1. A→ B : $\{N_a, A\}_{Kb}$
2. B→ A : $\{N_a, N_b\}_{Ka}$
3. A→ B : $\{N_b\}_{Kb}$

4a. A→ B : $\{\text{“}Transfer\ X1 \in from\ A\text{'}s\ account\ to\ Y\ 1\text{'}s\text{”}\}_{\langle N_a, N_b\rangle}$

4b. B→ A : $\{\text{“}Transfer\ X2 \in from\ B\text{'}s\ account\ to\ Y\ 2\text{'}s\text{”}\}_{\langle N_a, N_b\rangle}$

Fig. 4.1. NSPK++: the NSPK protocol terminated with the completion steps

This argument is after all not striking: even a proper evaluation of the "realism" of classical attacks found under DY would have required a proper cost/benefit analysis.

The research suggests that if in the real world an attacker can mount an attack that can be retaliated, then he may rationally opt for not attacking in the first place. In consequence, even a deployed protocol suffering an attack that can be retaliated may perhaps be kept in place.

The BUG threat model was demonstrated over the public-key Needham-Schroeder protocol [72]. Let us recast that analysis under the GAmodel. The protocol version studied here,which we address asNSPK++, is its original design terminated with the completion steps for reciprocal, authenticated money transfers (Figure 4.1). It can be seen that principal *A* issues a fresh nonce *Na* in message 1, which she sees back in message 2. Because message 1 is encrypted under *B*'s public key, the nonce was fresh and cryptanalysis cannot be broken by assumption, *A* learns that *B* acted at the other end of the network.Messages 2 and 3 give *B* the analogous information about *A* by means of nonce *Nb*. This protocol version is subject to Lowe's attack, as described in Figure 4.2. It can be seen how the attacker*C* masquerades as*A* with*B* to carry out an illegal money transfer at*B* (which intuitively is a bank) from*A*'s to *C*'s account. It is known that the problems originated with the confidentiality attack upon the nonce *Nb*. Another observation is that there is a second nonce whose confidentiality is violated—by *B*, not by *C*— in this scenario: it is *Na*. Although it is invented by*A*to be only sharedwith *C*, also *B* learns it. This does not seem to be an issue in the DY model, where all principals except *C* followed the protocol like soldiers follow orders.What one of them could do with a piece of information not meant for him therefore became uninteresting. To what extent this is appropriate to the current real world, where there often are various attackers with targets of their own, is at least questionable. Strictly speaking, *B*'s learning of *Na* is a new attack because it violates the confidentiality policy upon the nonces, which are later used to form a session key. It can be easily captured in the GA threat model, where more than one principal may act illegally at the same time for their own sake.

We are facing a new perspective of analysis. Principal *B* did not have to act to learn a nonce not meant for him, therefore this is named an *indeliberate attack*. To use a metaphor, *B* does not know which lock the key *Na* can open. This is not an issue in the GA threat model, where each principal just sends out anything he can send to anyone. Nevertheless, there are at least four methods to help *B* practically evaluate the potential of *Na* [74].

1. A→ C : $\{N_a, A\}_{Kc}$
1'. C(A) → B : $\{N_a, A\}_{Kb}$
2'. B→ A : $\{N_a, Nb\}_{K_a}$
2. C→ A : $\{N_a, Nb\}_{Ka}$
3. A→ C : $\{Nb\}_{Kc}$
3'. C(A) → B : $\{Nb\}_{Kb}$
4a'.C(A)→B:$\{\text{“}Transfer\ 1000€\ from\ A's\ account\ \text{to}\ C's\text{”}\}_{\langle N_a, N_b\rangle}$

Fig. 4.2. Lowe's attack to the NSPK++ protocol

4b'. B(C)→A: $\{\text{“}Transfer\ 2000€\ from\ C's\ account\ to\ B's\text{”}\}_{\langle N_a, N_b\rangle}$

Fig. 4.3. Retaliation attack following Lowe's attack

The natural consequence of *B*'s learning *Na* is the *retaliation attack* in Figure 4.3. Note that the first method indicated above gives *B* a reasonable set of target principals to try retaliation against, while the second one gives him a probabilistic answer originated from traffic analysis. However, the remaining two methods exactly tell him who the target for retaliation is.

It is worth remarking once more that a retaliation attack cannot be captured in the standard DY threat model, where all potential attackers merely collude to form a super-potent one. However, the GA model can support this notion.

An important finding that will be detailed below is that *B* learns *Na* before *C* learns *Nb* (Figure 4.2). This may lead to the unknown scenario that sees *B* steal money by step 4*b* from Figure 4.3 before *C* does it by step 4*a'* from Figure 4.2. The more quickly does *B* use any of the first three methods given above to evaluate *Na* and pinpoint *C*, the more realistic this scenario. Potentially, *B*'s illegal activity may even succeed before message 3 reaches *C* disclosing *Nb*. This attack will be addressed as *anticipation attack*.

Extending the Validation Method over the General Attacker

To perform the experiments: the SAT-based model checker SATMC, one of the AVISPA backends was used. This tool has successfully tackled the problem of determining whether the concurrent

execution of a finite number of sessions of a protocol enjoys certain security properties in spite of the DY attacker [75,76]. Leveraging on that work, we aim at relaxing the assumption of a single, super-potent attacker to specify the GA threat model, where principals can even compete each other. It was already stated above that the aim is to study novel protocol subtleties that go beyond the violation of standard security properties such as confidentiality and authentication. We are currently focusing on retaliation attacks and anticipation attacks. Also these notions can be recast into a model checking problem.

Basics of SAT-Based Model Checking

Let us outline the basic definitions and concepts underlying SAT-based model checking. The reader who is familiar with such concepts can skip this. Let us recall that a model checking problem can be stated as M |= G , where *M* is a labelled transition system modelling the initial state of the system and the behaviours of the principals (including their malicious activity) and *G* is an LTL formula expressing the security property to be checked.

The states of *M* are represented as sets of *facts* i.e. ground atomic formulas of a first-order language with sorts. If *S* is a state, then its facts are interpreted as the propositions holding in the state represented by *S*, all other facts being false in that state (closed-world assumption). A state is written down by the convenient syntax of a list of facts separated by the . symbol, as we shall see.

The transitions of *M* are represented as *set-rewriting rules* that define mappings between states. Each rule has a *label* expressing what the rule is there for: for example, the label $step_i$ is for a rule that formalises the *i*-th legal protocol step, the label overhear is for a rule whereby an attacker reads some traffic, and so on. Each rule label is parameterised by the rule variables or proper instances of them, and we will encounter a number of self-explaining rule labels below.

Let $(L \xrightarrow{p} R)$ be (an instance of) a rewriting rule and *S* be a set of facts. If $L \subseteq S$ then we say that ρ is applicable in *S* and that $S' = app_\rho(S) = (S \setminus L) \cup R$ is the state resulting from the execution of ρ in *S*. A *path* π is an alternating sequence of states and rules $S_0\rho_1S_1 \ldots S_{n-1}\rho_nS_n$ such that $S_i = app_{\rho i}(S_{i-1})$ (i.e. S_i is a state resulting from the execution of ρ_i in S_{i-1}), for $i = 1, \ldots, n$. Let *I* be the

initial state of the transition system; if $S0 \subseteq I$, then we say that the path is *initialised*. Let $\pi = S_0\rho_1S_1 \ldots S_{n-1}\rho_nS_n$ be a path. We define $\pi(i) = S_i$ and $\pi_i = S_ip_{i+1}S_{i+1} \ldots S_{n-1}\rho_nS_n$. Therefore, $\pi(i)$ and π_i are the *i*-th state of the path and the suffix of the path starting with the *i*-th state respectively. Also, it is assumed that paths have infinite length. This can be always obtained by adding stuttering transitions to the transition system.

The language of LTL used here has facts and equalities over ground terms as atomic propositions, the usual propositional connectives (namely, $\neg$, $\vee$) and the temporal operators **X** (next), **F** (eventually) and **O** (once). Let π be an initialised path of *M*, an LTL formula φ *is valid on* π, written $\pi \models \varphi$, if and only if $(\pi, 0) \models \varphi$, where $(\pi, i) \models \varphi$ (φ holds in π at time *i*) is inductively defined as:

$(\pi, i) \models f$ iff $f \in \pi(i)$ (*f* is a fact)

$(\pi, i) \models (t_1 = t_2)$ iff t_1 and t_2 are the same terms

$(\pi, i) \models \neg \varphi$ iff $(\pi, i) \not\models \varphi$

$(\pi, i) \models (\varphi 1 \vee \varphi 2)$ iff $(\pi, i) \models \varphi 1$ or $(\pi, i) \models \varphi 2$

$(\pi, i) \models \mathbf{X}\varphi$ iff $(\pi, i + 1) \models \varphi$

$(\pi, i) \models \mathbf{F}\varphi$ iff $\exists j \in [i,\infty).(\pi, j) \models \varphi$

$(\pi, i) \models \mathbf{O}\varphi$ iff $\exists j \in [0, i].(\pi, j) \models \varphi$

In the sequel we use $(\varphi 1 \wedge \varphi 2)$, $(\varphi 1 \Rightarrow \varphi 2)$ and $\mathbf{G}\varphi$ as abbreviations of $\neg(\neg\varphi 1 \vee \neg\varphi 2)$,

$(\neg\varphi 1 \vee \varphi 2)$ and $\neg\mathbf{F}\neg\varphi$ respectively.

Formalising the General Attacker

We conveniently adopt the IF language as it can specify inputs to the AVISPA backends and more specifically to SATMC, a successful SAT-based model checker [75] that will be used in the final validation phase. The following syntactical conventions are adopted in the sequel.

- Lower-case typewriter fonts, such as na, denote IF constants.
- Upper-case typewriter fonts, such as A, denote IF variables.
- Lower-case italics fonts of 0 arity, such as s indicating a session identifier, compactly denote IF terms.
- Lower-case italics fonts of positive arity, such as attack(a, v, s), denote metapredicates

- that aim at improving the readability of this manuscript, but in fact do not belong to the current IF formalisation.
- Upper-case italics fonts serve diverse purposes, as specified each time.

To specify the GA threat model, a number of new facts must be defined in our language. They are summarised with their informal meaning in Table1.

If *S* is a set of facts representing a state, then the state of principal *a* is represented by the facts of form $state_r(j, a, es, s)$ (called *state-facts*) and of form ak(*a*,*m*) occurring in *S*. It is assumed that for each session *s* and for each principal *a* there exists at most one fact of the form $state_r(j, a, es, s)$ in *S*. This does not prevent a principal from playing different roles in different sessions.

The dedicated account on the attacker's knowledge in DY must be extended as a general account on principals' knowledge in GA. In practice, what was the ik fact to represent the DY attacker knowledge is now replaced by ak, which has as an extra parameter the principal's identity whose knowledge is being defined. Incidentally, we conveniently write down the set of facts in a state by enumerating the facts and interleaving them a dot. Here is the definition of ak:

ak(a,m) . ak(a, k) $\xrightarrow{encrypt(VARS(a,k,m))}$ ak(a, $\{m\}_k$) . LHS

ak(a, $\{m\}_k$) . ak(a, $\bar{k}$) $\xrightarrow{decrypt(VARS(a,\bar{k},m))}$ ak(a,m) . LHS

ak(a,m_1) . ak(a, m_2) $\xrightarrow{pairing(VARS(a,m_1,m_2))}$ ak(a, $\langle m_1, m_2 \rangle$).LHS

ak(a, $\langle m_1, m_2 \rangle$) $\xrightarrow{decompose(VARS(a,m_1,m_2))}$ ak(a, m_1) . ak(a, m_2). LHS

where *VARS* (t_1, . . . , t_h) and *LHS* abbreviate in each rule, respectively, all the IF variables occurring in the IF terms represented by $t_1, \ldots, t_h$ and the set of facts occurring in the left hand side of the rule. Also, *k* and *k* are the inverse keys of one another.

Table 4.1. New facts and their informal meaning

Fact	Holds when
$state_r$(j, a, es, s)	Principal *a*, playing role *r*, is ready to execute step *j* in session *s* of the protocol, and has internal state *es*, which is a list of expressions affecting her future behaviour.
ak(*a*,*m*)	Principal *a* knows message *m*.
nt(*a*)	Principal *a* is not trustworthy.
c(*n*)	Term *n* is the current value of the counter used to issue fresh terms, and is incremented as s(*n*) every time a fresh term is issued.
msg(rs, b, a,m)	Principal *rs* has sent message *m* to principal *a* pretending to be principal *b*.
contains(*db*,*m*)	Message *m* is contained into set *db*. Sets are used, e.g., to share data between honest principals.
confidential(*m, g*)	Message *m* is a secret shared among the group of principals *g* (the set *g* is clearly populated through occurrences of contains(*g, a*) for each principal *a* intended to be in *g*).
transferred(rs,a, b, c,x,s)	*rs*, in the disguise of *a*, transferred *x* € from *a*'s account to *c* s at *b* (which intuitively is a bank) in session *s*.

Under the GA threat model any principal may behave as a DY attacker. We may nonetheless need to formalise protocols that encompass trusted third parties, or where a principal loses her trustworthiness due to a certain event. Reading through Table 1, it can be seen that our machinery features the nt(*a*) predicate, holding of a principal *a* that is not to be trusted. It may conveniently be used either statically, if added to the initial state of principals, or dynamically when introduced by rewriting rules. It is understood that if all principals but one are declared as trustworthy, then we are back to the DY threat model.

The fact c(*n*) holds of the current counter *n* used to generate fresh nonces. For example, c(0) holds in the initial state, and c(s(0)) after the generation of the first fresh nonce, which takes the value 0. More generally, if the fact c(na) holds, a fresh nonce na can be issued producing another state with counter c(s(na)).

The initial state of the system defines the initial knowledge and the statefacts of all principals involved in the considered protocol sessions. Its standard definition is omitted here but can be found elsewhere [77]. Appropriate rewriting rules specify the evolution of the system. Those for honest principals, which also serve to demonstrate the remaining facts enumerated in Table 1, are of the form:

msg(rs, b_1, a, m_1). $state_r$(i, a, es, s) $\xrightarrow{step_l(VARS(a,b_1,b_2,rs,es,es',m_1,m_2,s))}$ msg(a, a, b_2,m_2) . $state_r$(j, a, es', s) . ak(a, m_1). ak(a, m_2),

where l is the step label, i and j are integers and r is a protocol role (e.g., the NSPK++ has two roles Alice and Bob, also said Initiator and Responder roles). Rule 2 models the reception by a principal of a message and the principal's sending of the next message according to the protocol. More precisely, it states that if principal a, who is playing role r at step i with internal state es in session s of the protocol, has received message $m1$ supposedly from $b1$ (while the real sender is rs), then she can honestly send message $m2$ to $b2$. In doing so, a updates her internal state as $es\_$ and her knowledge accordingly, that is the new state registers the facts ak(a,$m1$) and ak(a,$m2$). Note that rule 2 may take slightly different forms depending on the protocol step it models. For example, if j = 1 and a sends the first message of the protocol, the fact msg(rs, $b1$, a,$m1$) does not appear in the left hand side of the rule, reflecting a's freedom to initiate the protocol at anytime. Similarly, for generating and sending a fresh term, c(N) is included in the left hand side of the rule, while c(s(N)) and ak(a, N) appear in the right hand side to express the incremented counter and the principal's learning the fresh term. A further variant is necessary when the step involves either a membership test or an update of a set of elements. In this case, facts of the form contains(db,m) must be properly defined. Facts such as confidential(m, g) or transferred(rs, b, a, c, x, s) added to the right hand side express respectively confidentiality for a group of principals, and a successful transfer of money.

To illustrate the specification of concrete protocol rules we consider two steps of the NSPK++ protocol. The transition in which B receives the first message of the protocol (supposedly) from A and replies with the second protocol message is modelled by the following rule:

msg(RS, A,B,$\{\langle A, NA\rangle\}_{\mathrm{KB}}$) . $state_{bob}$(1, B, [A, KA, KB, G], S) . c(NB) $\xrightarrow{step_2(B,A,RS,KA,KB,NA,NB,G,S)}$

msg(B, B, A,$\{\langle NA, NB\rangle\}_{\mathrm{KA}}$).$state_{bob}$ (3, B,[A, KA, KB, NA, NB, G],S).ak(B,$\{\langle A, NA\rangle\}_{\mathrm{KB}}$) . ak(B, NB) . contains(G, A) . contains(G, B) . confidential(NB, G) .c(s(NB)),

where G represents the group of principals that are allowed to share the freshly generated nonce NB. This also illustrates our different treatment of confidentiality with respect to DY's. While DY reduced confidentiality of a message to keeping the message confidential from the attacker, GA requires the original, subtler and unsimplified, definition of confidentiality: "confidentiality is the protection of information from disclosure to those not intended to receive it"[78]. For example, it regards the confidentiality of a message as compromised if ever anyone beyond its intended peers learns it. To provide another example of a protocol rule, the completion step $4b$ in which A receives and then executes a money transfer from B is modeled by two rules, one for B's sending and one for A's reception. Here is the latter:

msg(RS, B, A, $\{tr(B, C, X)\}_{\langle \mathrm{NA,NB}\rangle}$) . $state_{alice}$(4, A, [B, KA, KB, NA, NB, G], S)
$\xrightarrow{step_{4b_{rec}}(A,B,C,RS,KA,KB,NA,NB,X,G,S)}$
$state_{alice}$ (4, A, [B, KA, KB, NA, NB, G], S).ak(A, $\{tr(B, C, X)\}_{\langle \mathrm{NA,NB}\rangle}$) . ak(A, X) . transferred(RS, B, A, C, X, S)

The rule states the fact transferred(RS, B, A, C, X, S) to record that RS, who is not necessarily B, transferred X € from B's account to C's at A (which intuitively is a bank) in session S. This is not to be confused with $\{tr(B, C, X)\}_{\langle \mathrm{NA,NB}\rangle}$, which is a message expressing the request of a transfer.

The malicious behaviour of each principal C acting as a DY attacker can be specified by the following rules:

nt(C).ak(C, M).ak(C, A).ak(C, B)$\xrightarrow{fake(C,A,B,M)}$msg(C, A, B, M).LHS

nt(C) . msg(RS, A, B, M) $\xrightarrow{overhear(RS,A,B,C,M)}$ ak(C, M) . LHS

nt(C) . msg(RS, A, B, M) $\xrightarrow{intercept(RS,A,B,C,M)}$ ak(C, M) . nt(C)

Although the model outlined so far is accurate, it is not the most appropriate to perform automatic analysis efficiently. This is not a big issue for validating the NSPK++protocol in particular, but it is in

general. Themain problem is the specification of themalicious behaviour of principals. It allows for the forging ofmessages that will clearly not help to attack the protocol, as they do not correspond to the forms that the protocol prescribes. In other words, forging messages that no one will ever accept is of no use to any attacker. Efficiency of the analysis improves by adopting a refinedmodel ofmalicious activity, for example by introducing a forged message only if it conforms to one of the forms that belong to the protocol. More narrow-scoped, though realistic, impersonate rules detailed elsewhere [26] can easily be recast in the GA threat model. For example, the following impersonate rulemodels a principal C who, pretending to be A, sends the firstmessage of our example protocol to B, who is exactly waiting for a message of that form:

nt(C) . $state_{bob}$(2, B, [A, KA, KB, G], S).ak(C, A).ak(C, NA).ak(C, KB)

$$\xrightarrow{impersonate_2(C,A,B,KA,KB,NA,G,S)}$$

msg(C, A, B, $\{\langle A, NA\rangle\}_{\mathrm{KB}}$) . ak(C, $\{\langle A, NA\rangle\}_{\mathrm{KB}}$) . LHS

Formalising Protocol Properties under the General Attacker

One may expect that standard security properties such as confidentiality and authentication can be routinely specified and checked under the GA threat model using a model checker. Yet, confidentiality deserves particular consideration in what the GA model differs from the DY one.

In general, the confidentiality of a message m w.r.t. a group of principals g is guaranteed if and only if m is only known to principals in g. This property is clearly violated anytime a principal a outside g learns, for any reason, m. However, under DY, where all principals but the attacker meticulously follow the protocol, the violation is by design not considered so unless a is the attacker. There are many real scenarios—ranging from a betrayed person who publishes his/her partner's credit card details, to a scenario where a fired employee discloses sensitive information about its former employer—in which this violation is significant and therefore not negligible. In the GA model this confidentiality breach is not overlooked, as it can be

checked by the following meta-predicate formalising a violation of confidentiality:

$voc(a, m, g) = \mathbf{F}(\text{confidential}(m, g) \wedge \text{ak}(a,m) \wedge \neg \text{contains}(g, a))$

The negation of the meta-predicate given above corresponds to *G* in the model checking problem 1 and represents the confidentiality property.

The GA threat model paves the way to investigate subtler protocol properties than confidentiality and authentication. The retaliation is a common practice in the real world. (Even anyone who is most peaceful may react under attack—whether this is fortunate or unfortunate lies outside our focus.) Depending on who reacts against the attacker, retaliation can be named differently [16]: if it is the victim, then there is *direct retaliation*; if it is someone else, then there is *indirect retaliation*. Let *attack*(*c, a, b*) be a meta-predicate holding if and only if an attacker *c* has successfully attacked a victim *a* (being $a \neq c$ and $a \neq b$) with the (unaware) support of *b* (if any). Of course, "has successfully attacked" denotes the violation of the specific property that the protocol under analysis is supposed to achieve. Any violation of a protocol property should make the predicate hold, and therefore the predicate should be defined by cases. If we focus for simplicity on the didactic attack to the NSPK++ protocol seen above, which is an illegal money transfer, then the meta-predicate can be defined as:

$\text{attack}(c, a, b) = \text{transferred}(c, a, b, r, x, s) \wedge c \neq a$ (4)

Clearly, this definition can be extended to check other kinds of attacks. Because not all money transfers are illegal, the second conjunct in the formula is crucial: the illegal transfers are only those that are not requested by the account holder.

Direct and indirect retaliation can be respectively modelled as the following meta-predicates:

$\text{direct retaliation}(a,c) = \mathbf{F}(\text{attack}(c,a,b1) \wedge \mathbf{XF}\text{attack}(a, c, b2))$ (5)

$\text{indirect retaliation}(a,c,b) = \mathbf{F}(\text{attack}(c,a,b) \wedge \mathbf{XF}\text{attack}(b,c,a))$ (6)

It can be seen that the formula defined by Definition 4.5 is valid on those paths where an attacker hits a victim who hits the attacker back. Each attack can be carried out with the help of potentially different supporters *b*1 and *b*2. By contrast, Definition 6 of indirect retaliation shows that who hits the attacker back is not the victim but the

supporter, who perhaps realises what he has just done and decides to rebel against the attacker.

The definition of *anticipation attack* is deferred to the next subsection because it is best demonstrated upon the actual experiments.

Validating the NSPK++ Protocol under the General Attacker

The previous subs ection outlined the model checking problem in general, and described how to formalise the GA threat model for the validation of security protocols. For the sake of demonstration, it presented the formalisation of a step of the NSPK++ protocol. It concluded with the specification of protocol properties under the GA threat model.

Having digested the innovative aspects, deriving the full formalisation of the NSPK++ protocol under the General Attacker, which is omitted here, became an exercise. That formalisation was fed to SATMC, a state-of-the-art SAT-based model checker for security protocols, to carry out the first protocol validation experiments under GA. The details of SATMC appear elsewhere [76]. Its core is a procedure that automatically generates a propositional formula. The satisfying assignments of this formula, if any exist, correspond to counterexamples (i.e. execution traces of M that falsify G) of length bounded by some integer k, which can be iteratively deepened. Finding violations (of length k) on protocol properties boils down to solving propositional satisfiability problems. SATMC accomplishes this task by invoking state-of-the-art SAT solvers, which can handle satisfiability problems with hundreds of thousands of variables and clauses.

In running SATMC over the formalisation of the NSPK++ protocol, we considered the classical scenario in which a wants to talk with b in one session and with c in another session. Because the formalisation accounted for the new threat model, we were pleased to observe that it passed simple sanity checks: SATMC outputs the known confidentiality attack whereby c learns nb, and the known authentication attack whereby c impersonates a with b. As these are well known, they are omitted here. More importantly, the tool also reported what are our major findings: b's confidentiality attack upon na, and interesting retaliation attacks, which are detailed in the following. We argue that this is the first mechanised treatment of these subtle

properties, laying the ground for much more computer-assisted analysis of security protocols.

0: [step_1(a,c,ka,kc,na,nb,set_ac,1)]
1: [overhear(c,a,a,c,{na,a}kc)]
2: [decrypt(c,inv(kc),{na,a})]
3: [impersonate_2(c,a,b,ka,kb,na,set_ab,2)]
4: [step_2(b,a,c,ka,kb,na,nb,set_ab,2)]
5: [decrypt(b,inv(kb),{na,a})]

Fig.4. 4. Trace of NSPK++ featuring b's confidentiality attack

6: [impersonate_3(b,c,a,ka,kc,na,nb,set_ac,1)]
7: [step_3(a,c,b,ka,kc,na,nb,set_ac,1)]
8: [overhear(c,a,a,c,{nb}kc)]
9: [decrypt(c,inv(kc),{nb})]
10: [impersonate_3_rec(c,a,b,ka,kb,nb,set_ab)]
11: [step_3_rec(b,a,c,ka,kb,na,nb,set_ab,2)]
12: [impersonate_4a_rec(c,a,b,c,ka,kb,na,nb,set_ab,2)]
13: [step_4a_rec(b,a,c,c,ka,kb,na,nb,1K,set_ab,2)]
14: [impersonate_4b_rec(b,c,a,b,ka,kc,na,nb,set_ac,1)]
15: [step_4b_rec(a,c,b,b,ka,kc,na,nb,2K,set_ac,1)]

Fig. 4.5. Trace of NSPK++ (continuation) featuring b's indirect retaliation

Figure 4.4 reports the protocol trace that the tool outputs when checking a violation of confidentiality by the formula *voc*(A, M, G). Precisely, the trace is obtained by mildly polishing the partial-order plan returned by SATMC. Each trace element reports the label of the rule that fired, and hence the trace can be interpreted as the history of events leading to the confidentiality attack.

A full description of the trace requires a glimpse at the protocol formalization — each rule label in a trace element must be matched to the actual rule—but we will see that the trace can be automatically converted into a more user-friendly version. Element 0 means that principal a initiates the protocol with principal c. Elements 1, 2 and 3 show c's illegal activities respectively of getting the message, decrypting it and forging a well-formed one for principal b. Although c does not need in practice to overhear a message meant for him, element 1 is due to our monolithic formalisation of the acts of receiving a message and of sending out its protocol-prescribed reply. Therefore, for a principal to abuse a message, the principal must first overhear it.

Element 3 reminds that c's forging of the message for b counts as an attempt to impersonate a. The last two elements signal b's legal participation in the protocol by receiving the message meant for him, and his subsequent deciphering of the nonce. SATMC now returns because *voc*(b, na, set_ac) holds (where the set set_ac = {a, c}).

To illustrate the detection of a retaliation attack, SATMC can be launched on a significant property such as *indirect retaliation*(A, B, C). It returns a trace that continues as in Figure 4.5 the one seen in Figure 4.4. The impersonate rules show that both b and c are acting illegally in this trace, a development that has never been observed by previous analyses under DY. Elements 6 and 7 show that b is impersonating c with a, who naively replies to c. The next three elements confirm c's attempt at fooling b, who legally replies as element 11 indicates. Now c can finalise his attack as in elements 12 and 13. The latter element indicates the firing of rule 4*a*, which makes *attack*(c, a, b) hold. The last two elements witness b's retaliation attack by making *attack*(b, c, a) hold. Therefore, by Definition 4.6, the property *indirect retaliation*(a, c, b) holds, indicating that the tool reports the retaliation attack described above (Figure 3.3).

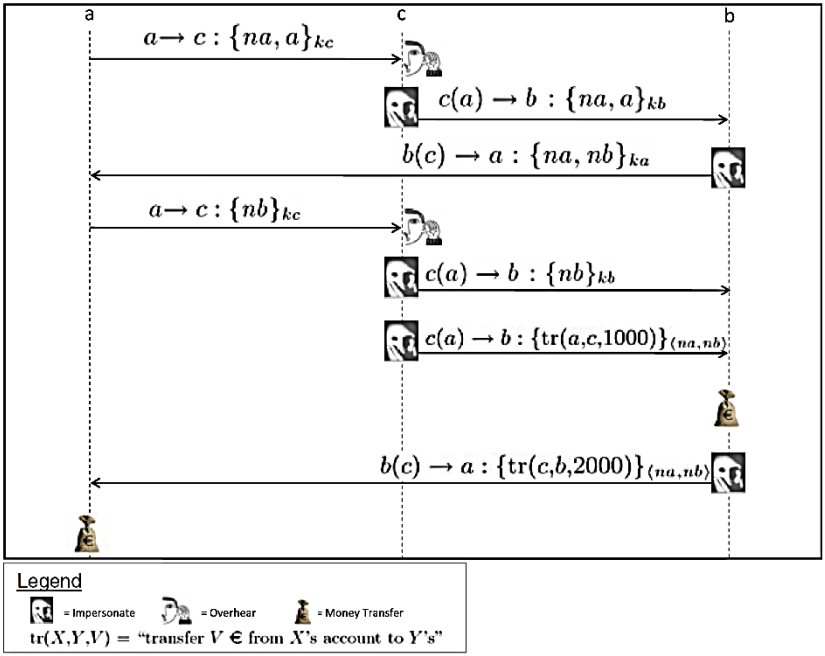

Fig. 4.6. Graphics of b's indirect retaliation in NSPK++

A graphical illustration of this retaliation attack is in Figure 6, and can be easily built. First, we run a procedure to transform the output of the tool into a more readable and intuitive version. Then, we coherently associate the significant phrases of this version to graphical elements, and hence build the image. The behaviours and interactions of the principals can be observed by looking at the figure from top to bottom. The overhear and impersonate icons emphasise the significant number of illegal steps taken by both principals c and b).

We run the tool repeatedly on the protocol model, each time adjusting the property to check. In particular, we tried an alternative definition of indirect retaliation where the **X** operator was left out. The tool returned a trace leading to a state where both attacks held, as if retaliation did not need be triggered by another attack. This called for more attention at the trace.

In consequence, we observed from element 3 in Figure 4.4 that c initialises his attack very early—precisely, with his impersonation of a based upon the repetition to b of a's nonce na. This means that b learns

na when c has not yet learned nb and hence cannot attack yet. We thus defined the self-explaining fact nonce_leak(*c, a, b*). In the GA threat model, it is plausible that b exploits his knowledge before c does. This can be interpreted as a scenario in which a potential victim realises that he is going to be attacked, and therefore reacts successfully before being actually attacked. We name such a successful reaction *anticipation attack* and define it as a meta-predicate below. For the sake of efficiency, we decided to add a rule that introduces the fact nonce leak(C, A, B) following c's attack initialisation:

confidential(NA, G) . ak(B, NA) . ¬ contains(G, B) .

ak(B, KB^{-1}) . msg(C, A, B, $\{\langle A, NA\rangle\}_{\mathrm{KB}}$) $\xrightarrow{i\_got\_you(C,A,B,NA,G)}$

nonce_leak(C, A, B)

A meta-predicate *attack init*(*c, a, b*) must be introduced to formalise some *c*'s act of initialising an attack, which is yet to be carried out, while interleaving sessions with some *a* and *b*. In the same fashion of Definition 4 of *attack*(*c, a, b*), we provide a definition that is appropriate for the attack under analysis, and corresponds to the nonce leak:

attack_init(c, a, b) = nonce_leak(c, a, b)

The definition of anticipation attack can be given now. It insists that an attack initiated by someone, such as *c*, is followed by an actual attack carried out by someone else, such as *b*:

anticipation_attack(c, a, b) = **F**(attack_init(c, a, b) ∧
XF*attack*(*b, c, a*)) (7)

SATMC validated our intuition.When run on *anticipation attack*(C, A, B) as a goal, the tool produced the partial order plan reported in Figure 7. Element 6 in the trace indicates that c leaked a nonce created by a by sending it to b. So, the meta-predicate *attack_init*(c, a, b) holds.Moreover, the last two elements witness b's anticipation attack by making *attack*(b, c, a) hold. Therefore, by Definition 4.7, we have that *anticipation_attack*(c, a, b) is true, which signifies that the tool reports the anticipation attack described above (Figure 4.8).

0: [step_1(a,c,ka,kc,na,nb,set_ac,1)]
1: [overhear(c,a,a,c,{na,a}kc)]
2: [decrypt(c,inv(kc),{na,a})]
3: [impersonate_2(c,a,b,ka,kb,na,set_ab,2)]
4: [overhear(b,c,a,b,{na,a}kb)]
5: [decrypt(b,inv(kb),{na,a})]
6: [i_got_you(c,a,b,na,set_ac)]
7: [impersonate_3(b,c,a,ka,kc,na,nb,set_ac,1)]
8: [step_3(a,c,b,ka,kc,na,nb,set_ac,1)]
9: [impersonate_4b_rec(b,c,a,b,ka,kc,na,nb,set_ac,1)]
10: [step_4b_rec(a,c,b,b,ka,kc,na,nb,2K,set_ac,1)]

Figure. 4.7. Trace of NSPK++ featuring b's anticipation attack

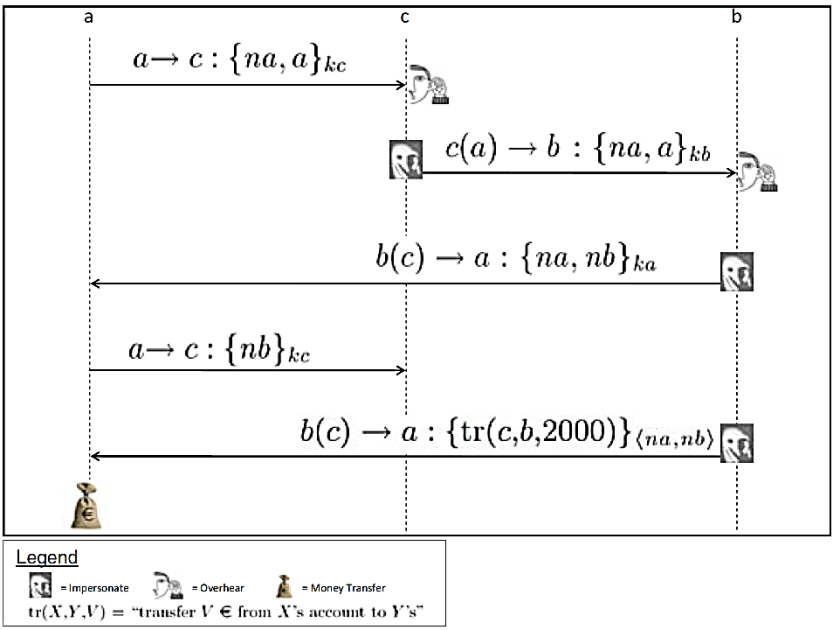

Figure. 4.8. Graphics of b's anticipation attack in NSPK++

Various experiments produced, that SATMC reported a trace describing the following scenario. In a session, a discloses to c her

nonce generated for b. In another session, a discloses to b another nonce of hers generated for c. In consequence, both b and c become capable of attacking each other with a's support if needed. Moreover, b and c may be initially unaware of each other's capability of attack. This reflects the real-world situation in which someone creates strife in a couple that starts fighting.

Conclusions

The General Attacker threat model seems most appropriate to the present social/technological setting. Reasoning that was impossible under DY can now be carried out, highlighting protocol niceties that are routinely overseen.

Retaliation teaches us that we can perhaps live with flawed protocols. We are used to go back to design when a protocol is found flawed, even if already deployed. However, an attack that can be retaliated may in practice convince an attacker to refrain from attacking in the first place. If the "cost" of attacking overdoes its "benefits" then the attacker will not be carried out. Retaliation makes that precondition hold.

Anticipation teaches us to ponder the entire sequence of events underlying an attack. An attack typically is an interleaving of legal and illegal steps rather than a single illegal action. Therefore, we may face a scenario, unreported so far, where a principal mounts an attack by successfully exploiting for his own sake the illegal activity initiated but not yet finalised by someone else. This is routine for the present hackers' community.

Tasks for laboratory work №4.

1. According to given in Table 4.2 variant of the security protocols define its properties.
2. According to given in Table 4.2 variant use the threat model and automatically validate a protocol under the new threats, so that retaliation and anticipation attacks can automatically be found.

Table 4.2. Variants

№	Protocol	Threat model
1	S/MIME	Dolev-Yao
2	VPN	Rational Attacker

3	IPSec	General Attacker
4	TLS	Dolev-Yao
5	SSL	Rational Attacker
6	HTTPS	General Attacker
7	PGP	Dolev-Yao
8	S-HTTP	Rational Attacker
9	KERBEROS	General Attacker
10	SET	Rational Attacker

Requirements to the report

The report should consists of:

- title sheet;
- the aim and the task of the laboratory work;
- the list of the defined properties of the security protocols;
- results of the usage of the threat model and validation of the protocol under the new threats with the demonstration that retaliation and anticipation attacks can automatically be found;
- conclusions.

Advancement questions

1. Can Dolev-Yao be considered the standard threat model to study security protocols and why?
2. Can Dolev-Yao attacker control the entire network without perform cryptanalysis and how?
3. Can the relation attack be captured in the standard Dolev-Yao threat mode and why?
4. What tool is able to successfully tackle the problem of determining whether the concurrent execution of a finite number of sessions of a protocol enjoys certain security properties ?
5. What are the main syntactical conventions are adopted to Formalize the General Attacker?
6. How the dedicated account on the attacker's knowledge in Dolev-Yao must be extended?
7. What the initial state of system defines?
8. What are the main definitions of SAT-based model?
9. How to define properties of security protocols?

10. How to validate a protocol under the new threats, so that retaliation and anticipation attacks can automatically be found?

FORMAL AND INTELLECTUAL METHODS FOR SYSTEM SECURITY AND RESILIENCE
3 FORMAL AND INTELLECTUAL METHODS FOR SYSTEM SECURITY AND RESILIENCE

3.1 Seminar №1. Formal Goal-Oriented Development of Resilient MAS in Event-B

The aim and the task of the laboratory work

The aim of this laboratory work is to get acquainted with a formal goal-oriented approach to development of resilient multi agent system.

Task of the work:

- to define goals in Event-B and ensure goal reachability by refinement.
- To defined a set of modelling and refinement patterns that describe generic solutions common to formal modelling of multi agent system
- Use the rigorous modelling of the impact of agent failures on goal achieving I order to build a dynamic goal reallocation mechanism that guarantees system resilience in presence of agent failures

Preparation for laboratory work

- to clarify the aims and objectives;
- to study theoretical material given in the description.

Theoretical material

Introduction

Goal-Oriented Development [79,80] has been recognised as an useful framework for structuring and specifying complex system requirements. In goal-oriented development, the system requirements are defined in terms of goals - the functional and non-functional objectives that a system should achieve. Often changes in system operational environment, e.g., caused by failures of agents - independent system components of various types - might hinder achieving the desired goals. Hence, to ensure system resilience [81], i.e., guarantee its dependability in spite of the changes, we need formally verify reachability of the targeted goals. Traditionally, such a verification is undertaken by abstracting implementation up to

requirements level and model-checking satisfiability of goals. However, such an approach suffers from a state explosion that is especially prohibitive for such applications as multi-robotic systems [82].

Let us consider a formal development approach that ensures goal reachability "by construction". It is based on refinement in Event-B. Event-B [83] is a formal top-down development approach to correct-by-construction system development. The main development technique - refinement - allows us to ensure that a concrete specification preserves globally observable behaviour and properties of abstract specification. Verification of each refinement step is done by proofs. The Rodin platform [84] automates modelling and verification in Event-B. Currently Event-B is actively used within EU project DEPLOY [85] to model dependable systems from various domains.

The goal-oriented development by defining a set of specification and refinement patterns is formalised. The formalisation reflects the main concepts of the goal- oriented engineering. In particular, we demonstrate how to define system goals at different levels of abstraction and guarantee goal reachability while specifying collaborative agent behaviour. Moreover, we propose refinement patterns that allow the system to dynamically reallocate goals from failed agents to healthy ones and per se, guarantee resilience. A development of an autonomous multi- robotic system illustrates application of the proposed patterns.

Formal Modelling and Refinement in Event B

Let us consider formal framework - Event-B. The Event-B formalism is an extension of the B Method [86]. It is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event-B has been specifically designed to model and reason about parallel, distributed and reactive systems.

Modelling in Event-B

In Event-B, a system model is specified using the notion of an *abstract state machine* [87]. An abstract state machine encapsulates the system state represented as a collection of model variables, and defines operations on this state, i.e., it describes the dynamic *behaviour*

of the modelled system. A machine may also have the accompanying component, called *context*. A context might include user- defined carrier sets, constants and their properties, which are given as a list of model axioms. In Event-B, the variables are strongly typed by the constraining predicates called **invariants**. Moreover, the invariants specify important properties that should be preserved during the system execution.

The dynamic behaviour of the system is defined by the set of atomic events. Generally, an event can be defined as follows:

evt ≙ any *ul* where *g* then *S* end

where *vl* is a list of new local variables (parameters), *g* is the event **guard**, and *s* is the event **action**. The guard is a state predicate that defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks. In general, the action of an event is a parallel composition of deterministic or non-deterministic assignments.

Event-B Refinement

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract system specification that non-deterministically models the most essential functional requirements. In a sequence of refinement steps we gradually reduce non-determinism and introduce detailed design decisions. In particular, we can replace abstract variables by their concrete counterparts, i.e., perform data refinement. In this case, the invariant of the refined machine formally defines the relationship between the abstract and concrete variables. Via such a *gluing* invariant we establish a correspondence between the state spaces of the refined and the abstract machines.

Often a refinement step introduces new events and variables into the abstract specification. The new events correspond to the stuttering steps that are not visible at the abstract level, i.e., they refine implicit *skip*. To guarantee that the refined specification preserves the global behaviour of the abstract machine, we should demonstrate that the newly introduced events *converge*. To prove it, we need to define a

variant - an expression over a finite subset of natural numbers - and show that the execution of new events decreases it. Sometimes, convergence of an event cannot be proved due to a high level of non-determinism. Then the event obtains the status *anticipated.* This obliges the designer to prove at some later refinement step, that the event indeed converges.

Each refinement step requires to verify a number of proof obligations that ensure that the refined specification adheres to its abstract counterpart [87]. The verification efforts, in particular, automatic generation and proving of the required proof obligations, are significantly facilitated by the Rodin platform [84].

Refinement and proof-based verification of Event-B offers the designers a scalable support for the development of such complex systems as multi-agent systems (MAS). MAS are decentralised distributed systems composed of agents asynchronously communicating with each other. Agents are computer programs acting autonomously on behalf of a person or organisation, while coordinating their activities by communication [88]. MAS are increasingly used in various critical applications such as factories, hospitals, rescue operations in disaster areas, etc.

A Formal View of Goal-Oriented Multi-Agent System. Patterns for Goal-Oriented Development

The goal-oriented engineering facilitates structuring complex system requirements in terms of *goals* - objectives that the system should meet [80]. In this subsection we focus on modelling functional goals, i.e., the goals defining objectives of the services that the system should deliver. We propose a number of *specification and refinement patterns* that interpret essential activities of goal-oriented engineering in terms of Event-B refinement.

A pattern in Event-B is an abstract machine that defines a generic modelling solution that can be reused in similar developments via instantiation. Usually, an Event-B pattern contains abstract types, constants and variables. The context of such a model constraints the instantiation by defining the properties that should be satisfied by concrete instantiations of abstract data structures. The invariant properties of a pattern, once proven, remain valid for all instantiations.

The aim of defining a pattern is to capture experience gained in modelling a certain problem. To illustrate how patterns are defined, let us now present a pattern that allows the designers to explicitly define goals while modelling a system in Event-B. We call it *Abstract Goal Modelling Pattern.*

Abstract Goal Modelling Pattern

Let *GST ATE* be an abstract type defining the system state space[3]. Moreover, let *Goal* be a non-empty proper subset of *GST AT E* that abstractly defines the given system goals. We say that the system has achieved the desired goals if its current state belongs to *Goal.* Both *GST ATE* and *Goal* are the abstract types. Together with their properties they are defined in the model context as follows:

$Goal \neq \varnothing$ and $Goal \subset GST\ ATE.$

Let us note that *GSTATE* and *Goal* are generic parameters of the initial pattern. During a system development, we should supply their concrete instantiations that satisfy the properties shown above.

While modelling a system in Event-B, we should ensure that the system under development achieves the desired goal. We can formally express this by requiring that the system terminates in a state belong to *Goal*. The machine M_AGM is defined according to the *Abstract Goal Modelling Pattern*:

Machine M_AGM
Variables $gstate$
Invariants $gstate \in GSTATE$
Events
Initialisation $\widehat{=}$
begin
$gstate :\in GSTATE \setminus Goal$
end
Reaching_Goal $\widehat{=}$
status $anticipated$
when
$gstate \in GSTATE \setminus Goal$
then
$gstate :\in GSTATE$
end
end

The dynamic behaviour of the system is abstractly modelled by the event **Reaching_Goal**. The system terminates when **Reaching_Goal** becomes disable, i.e., when a state satisfying *Goal* is reached.

The event **Reaching_Goal** has the status *anticipated***.** Hence, in the machine **M_AGM** goal reachability is postulated rather than proved. However, it also obliges us to prove (at some refinement step) that the event or its refinements converge. Therefore, while refining a concrete specification defined according to *Abstract Goal Modelling Pattern*, we will be forced to prove goal reachability.

Let us assume that we have a collection of Event-B patterns: P_1, P_2, ..., P_n that refine each other in the following way:

P_1 is refined by P_2 ... is refined by P_n**.**

Such a refinement chain expresses a generic development by refinement. Abstract data structures of all the involved patterns become generic parameters of the development. Each pattern abstractly defines a solution for specifying a certain modelling aspect. Therefore, each refinement step has a rationale behind it - its meta-level description. We use it to formulate modelling aspects that the refinement transformation aims at defining. The result of refinement transformation is called a refinement pattern.

Next we propose several refinement patterns that allow us to implement the ideas of goal-oriented engineering in Event-B refinement. We start from defining *Goal Decomposition Pattern*.

Goal Decomposition Pattern

The main idea of goal-oriented development is to decompose the high-level system goals into a set of subgoals. This is an iterative process that aims at building the hierarchy of system goals. Essentially, subgoals define intermediate stages of the process of achieving the main goal.

The purpose of *Goal Decomposition Pattern* is to explicitly model subgoals in the system specification. While defining this pattern, we should ensure that high- level goals remain achievable. Hence the refinement pattern should reflect the relation between the high-level goals and subgoals. Moreover, it should ensure that high-level goal reachability is preserved and can be defined via reachability of lower-layer subgoals.

In this subsection we assume that subgoals are independent of each other. This means that reachability of any subgoal does not affect reachability of another one. Moreover, while a certain subgoal is reached, it remains reached, i.e., the system always progresses towards achieving its goals. Formally, it can be expressed as a stability property with respect to some state predicate P:

$$Stable(P) \Leftrightarrow \text{“once } P \text{ becomes true it remains true”}.$$

In Event-B, stability properties can be easily expressed by introducing auxiliary variables for storing the previous value of the state and then formulating stability properties as the invariant properties of the form:

$$P(prev\ state) = TRUE \Rightarrow P(state) = TRUE.$$

To express a goal decomposition in terms of Event-B, let us define a corresponding refinement pattern. We present it by the machine M_GD. The new pattern allows us to introduce a number of subgoals into our system model and express their reachability. Moreover, the refinement relation between patterns allows us to express reachability of the main goal via reachability of its subgoals.

Let us assume for simplicity, that system goal *Goal* is achieved by reaching three subgoals. The subgoals are defined as corresponding

variables of the M_GD machine: *Subgoali, Subgoal*$_2$, and *Subgoal*$_3$. The goal independence assumption allows us to partition high-level goal state space *GST ATE* into three non-empty subsets: *SGST AT E1, SGST ATE*2, *SGST ATE3.* We define the subgoals as follows:

$$Subgoali \neq \varnothing \ \textbf{and}\ Subgoali \subset SGSTATEi,\ i \in \textbf{1..3.}$$

To establish a relationship between the new state spaces *SG_STATEi, i* $\in$ 1..3, of the M_GD machine and the abstract state space of M_AGM machine we define the following function:

$$State\_map \in SG\_STATE1\ x\ SG\_STATE2\ \textbf{x}\ SG\_STATE3 \rightarrow GSTATE,$$

where $\rightarrow$ designates a bijection function. Essentially it partitions the original goal state space into three independent parts.

To postulate that the main goal is reached if and only if all three subgoals are reached, we add an axiom into the context of the M_GD machine:

$$\forall sg1, sg2, sg3 \cdot sg1 \in Subgoal_1 \wedge Subgoal_2 \wedge sg3 \in Subgoal_3 \Leftrightarrow$$

$$State\_map(sg1 \mapsto sg2 \mapsto sg3) \in Goal$$

Refinement performed according to the *Goal Decomposition Pattern* is an example of the Event-B data refinement. We replace the abstract variable *gstatei* with the new variables *gstatei* $\in$ *SG_STATEi, i* $\in$ 1..3. The new variables model the state of the corresponding subgoals. The following gluing invariant allows us to prove data refinement:

$$gstate = State\_map(gstate1 \mapsto gstate2 \mapsto gstate3)$$

Essentially the M_GD machine decomposes the Reaching_Goal event of the M_AGM machine into three similar events Reaching_SubGoali, $i \in$ 1..3:

Machine M_GD
Reaching_SubGoal$_i$ $\widehat{=}$ refines Reaching_Goal
status $anticipated$
when
$gstate_i \in SG\_STATEi \setminus Subgoal_i$
then
$gstate_i :\in SG\_STATEi$
end
…

Let us observe that we can easily verify that the following stability property holds for the pattern M_GD:

$Stable(gstate_1 \in Subgoal_1) \wedge Stable(gstate_2 \in Subgoal_2) \wedge$
$Stable(gstate_3 \in Subgoal_3)$

The proposed *Goal Decomposition Pattern* can be repeatedly used to refine subgoals into the subgoals of finer granularity until the desired level of details is reached.

Agent Modelling Pattern

The elaborated *Abstract Goal Modelling* and *Goal Decomposition* patterns allow us to specify the system goal(s) at different levels of abstraction. In multi-agent systems, (sub)goals are usually achieved by system agents. Agents are independent entities that are capable of performing certain tasks. In general, the system might have several types of agents that are distinguished by the type of tasks that they are capable of performing. The next refinement pattern - *Agent Modelling Pattern* - allows us to model agents and associate them with goals.

We introduce the set *AGENTS* that abstractly defines the set of system agents. In this refinement pattern we also introduce a concept of agent *eligibility.* An agent is *eligible* if it is capable of achieving a certain task (subgoal). We define the non-empty sets *EL_AG1*, *EL_AG2,* and *EL_AG3* of the agents eligible to achieve each particular subgoal.

Agent might fail while trying to achieve a certain subgoal. Then it is removed from the dynamic set of the eligible agents represented by the variable *eligi: eligi* $\subseteq$ *ELAGi, i* $\in$ 1..3.

A goal is achieved if there is at least one eligible agent associated with it. This is formulated as the corresponding invariant property in the pattern:

$$elig_1 \neq \varnothing and\ elig_2 \neq \varnothing and\ elig_3 \neq \varnothing$$

The dynamic part of the *Agent Modelling Pattern* is defined in the machine M_AM. Since we assumed that the agents can fail, the goal assigned to the failed agent cannot be reached. To reflect this assumption in our model, we refine the abstract event Reaching_SubGoali by two events Successful_Reaching_SubGoali and Failed_Reaching_SubGoali, $i \in 1..3$, which respectively model successful and unsuccessful reaching of the subgoal by some eligible agent:

Machine M_AM

Successful_Reaching_SubGoal$_1$ $\widehat{=}$ refines Reaching_SubGoal$_1$

status *convergent*

any ag

when

$gstate1 \in SG\_STATE1 \setminus Subgoal1 \wedge ag \in elig_1$

then

$gstate1 :\in Subgoal1$

end

Failed_Reaching_SubGoal$_1$ $\widehat{=}$ refines Reaching_SubGoal$_1$

status *convergent*

any ag

when

$gstate1 \in SG\_STATE1 \setminus Subgoal1 \wedge ag \in elig1 \wedge card(elig1) > 1$

then

$gstate1 :\in SG\_STATE1 \setminus Subgoal1$

$elig1 := elig1 \setminus \{ag\}$

end

...

In the guard of the event Failed_Reaching_SubGoali we restrict possible agent failures by postulating that at least one agent associated with the subgoal remains operational: *card(eligi) > 1, i* ∈ 1..3. This assumption allows us to change the event status from anticipated to convergent. In other words, we are now able to prove that, for each subgoal, the process of reaching it eventually terminates. To prove the convergence we define the following variant expression:

$$\text{card}(\text{elig}_1) + \text{card}(\text{elig}_2) + \text{card}(\text{elig}_3).$$

When an agent fails, it is removed from a corresponding set of eligible agents *eligi.* This in turn decreases the value of *card(eligi)* and consequently the whole variant expression. On the other hand, when an agent succeeds in reaching the goal, all the events become disabled, thus ensuring system termination as well.

In practice, the constraint to have at least one operational agent associated with our model can be validated by probabilistic modelling of goal reachability. Let us also note that for multi-robotic systems with many homogeneous agents this constraint is usually satisfied.

Agent Refinement Pattern

Above we have defined the notion of agent eligibility quite abstractly. We establish the relationship between subgoals (tasks) and agents that are capable of achieving them. The last refinement pattern, *Agent Refinement Pattern,* aims at unfolding the notion of agent eligibility. Here we define the agent eligibility by introducing agent attributes - *agent types* and *agent statuses.* An eligible agent will be an operational agent that belongs to a particular agent type.

We define an enumerated set of agent types *AG_TYPE = {TYPE1, TYPE*2, *TYPE3}* and establish the correspondence between abstract sets of eligible agents and the corresponding agent types by the following axioms:

$\forall ag \cdot ag \in EL\_AGi \Leftrightarrow atype(ag) = TYPEi, i \in 1..3$

Fig.5.1. An agent is eligible to perform a certain subgoal if it has the type associated with this subgoal.

An agent might be operational or failed. To model the notion of agent status we define an enumerated set *AGSTATUS = {OK, KO},* where constants *OK* and *KO* designate operational and failed agents correspondingly.

Below we present an excerpt from the dynamic part of the *Agent Refinement Pattern* - the machine M_AR. We add a new variable *astatus* to store the dynamic status of each agent:

$astatus \in AGENTS \rightarrow AG\_STATUS.$

Moreover, we data refine the variables *eligi.* The following gluing invariants relate them with the concrete sets:

$elig_i = \{a \mid a \in AGENTS \wedge atype(a) = TYPEi \wedge$

$astatus(a) = OK\}, i \in 1..3$

In our case, the dynamic set of agents eligible to perform a certain subgoal becomes a set of active agents of the particular type. The event Failed_Reaching_SubGoal$_1$ is now refined to take into account the concrete definition of agent eligibility. The event also updates the status of the failed agent.

Successful_Reaching_SubGoal$_1$ $\widehat{=}$ refines Successful_Reaching_SubGoal$_1$
  any ag
  when
    $gstate1 \in SG\_STATE1 \setminus Subgoal1 \wedge astatus(ag) = OK \wedge atype(ag) = TYPE1$
  then
    $gstate1 :\in Subgoal_1$
  end
Failed_Reaching_SubGoal$_1$ $\widehat{=}$ refines Failed_Reaching_SubGoal$_1$
  any ag
  when
    $gstate1 \in SG\_STATE1 \setminus Subgoal1 \wedge astatus(ag) = OK \wedge atype(ag) = TYPE1 \wedge$
    $card(\{a | a \in AGENTS \wedge atype(a) = TYPE1 \wedge astatus(a) = OK\}) > 1$
  then
    $gstate1 :\in SG\_STATE1 \setminus Subgoal_1 \parallel astatus(ag) := KO$
  end

As mentioned above, to prove the defined goal reachability property, we had to make the assumptions related to agent reliability, i.e., assume that some agents remain operational to successfully complete the goal achieving process. To validate this assumption, we can employ quantitative assessment probabilistic model checking techniques.

To enable probabilistic analysis of Event-B models in the probabilistic model checker PRISM, we can rely on the continuous-time probabilistic extension of the Event-B framework. The idea of this approach is as follows. We annotate actions of all model events with real-valued rates (e.g.,failure rate, service rate) and then transform such a probabilistically augmented Event-B specification into a continuous-time Markov chain, which we can represent in PRISM. Then we can assess the probability of achieving the goal as well as to compare several alternative system configurations.

The resilience-explicit goal-oriented refinement approach presented above allowed us to identify the key concepts required for formal development of resilient MAS. It has inspired as to propose a conceptual framework for goal-oriented reasoning about resilient MAS

that puts a specific emphasis of rigorous definition of of system reconfigurability.

Case Study: a Multi-Robotic System. A Case Study Description

As a case study we consider a multi-robotic system. The goal of the system is to coordinate identical robots to get a certain area cleaned. The area is divided into several zones, which can be further divided into a number of sectors. Each zone has a base station - a static computing and communicating device - that coordinates the cleaning of the zone. In its turn, each base station supervises a number of robots by assigning cleaning tasks to them.

A robot is an autonomous electro-mechanical device - a special kind of a rover that can move and clean. The base station may assign a robot a sector a certain area in the zone - to clean. As soon as the robot receives a new cleaning task, it autonomously travels to this area and starts to clean it. After successfully completing its mission, it returns back to the base station to receive a new order. The base station keeps track of the cleaned sectors. A robot may fail to clean the assigned sector. In that case, the base station assigns another robot to perform this task. To ensure that the whole area is eventually cleaned, each base station in its turn should ensure that its zone is eventually cleaned.

The system should function autonomously, i.e., without human intervention. Such kind of systems are often deployed in hazardous areas (nuclear power plants, disaster areas, mine fields, etc.). Hence guaranteeing system resilience is an important requirement. Therefore, we should formally demonstrate that the system goal is achievable despite possible robot failures.

Pattern-Driven Refinement of a Multi-Robotic System

Let us consider the case study that describes the formal development of a multi-robotic system in Event-B. The development is concluded via instantiation of the proposed patterns, with the goal decomposition pattern being applied twice in a row.

Abstract model. The initial model defined by the machine MRS_Abs specifies the behaviour of a multi-robotic system according to the *Abstract Goal Modelling Pattern.* We apply this pattern by

instantiating abstract variables with the concrete values and specifying events that model system behaviour.

The state space of the initial model is defined by the type *BOOL*. The value TRUE corresponds to the situation when the desired goal is achieved (i.e., the whole territory is cleaned), while FALSE represents the opposite situation.

Similarly to the pattern machine M_AGM, the machine MRS_Abs contains an event, CleaningTerritory, that models system behaviour. It abstractly represents the process of cleaning the territory, where a variable *completed € BOOL* models the current state of the system goal. This event is constructed according to the pattern event Reaching_Goal by taking all the instantiations into account, as shown below:

Machine AbsMRS
Variables $completed$
Invariants $completed \in BOOL$
Events
...
CleaningTerritory $\widehat{=}$
status $anticipated$
when
$completed = FALSE$
then
$completed :\in BOOL$
end

The system continues its execution until the whole territory is cleaned, i.e., as long as *completed* stays FALSE. At this level of abstraction, the event CleaningTerritory has the *anticipated* status. In other words, similarly to the abstract pattern, we delay the proof that the event eventually converges to subsequent refinements. It is easy to see that the machine AbsMRS is an instantiation of the pattern machine M_AGM, where the abstract type *GSTATE* its replaced with *BOOL*, the constant *Goal* is instantiated with a singleton set {TRUE}, and the variable *gstate* is renamed into *completed.*

First refinement. The initial model specifies system behaviour in a highly abstract way. It models the process of cleaning the whole territory. The goal of the first refinement is to model the cleaning of the

territory zones. Refinement is performed according to the *Goal Decomposition Pattern*.

In the first refinement step resulting in the machine MRS_Ref1, we augment our model with representation of subgoals. The whole territory is divided into *n* zones, $n \in$ N and $n >= 1$. We associate the notion of a *subgoal* with the process of *cleaning a particular zone.* Thus a subgoal is achieved when the corresponding zone is cleaned. A new variable *zone-completed* represents the current subgoal status for every zone. The value TRUE corresponds to the situation when the certain zone is cleaned:

$$zone\_completed \in 1..n \rightarrow BOOL.$$

The refined model MRS_Ref1 is built as an instantiation of the *Goal Decomposition Pattern* machine M_GD, where the subgoal states are defined as elements of the variable *zone_comvleted,* i.e., $gstate_i = zone\_completed(i), for\ i \in 1..n.$

This observation suggests the following gluing invariant between the initial and the refined models:

$$comleted = TRUE \Leftrightarrow zone\_completed[1..n] = \{TRUE\}$$

The invariant can be understood as follows: the territory is considered to be cleaned if and only if its every zone is cleaned.

The pattern events Reaching.SubGoal$_i$ correspond to a single event Cleaning Zone.

Machine MRS_Ref1
CleaningZone $\widehat{=}$ refines CleaningTerritory
status $anticipated$
any $zone, zone\_result$
when
$zone \in 1..n \wedge zone\_completed(zone) = FALSE \wedge$
$zone\_result \in BOOL$
then
$zone\_completed(zone) := zone\_result$
end

Second refinement. In the development of a multi-robotic system we should apply the goal decomposition pattern twice, until we reach the level of "primitive" goals, i.e., the goals for which we define the classes of agents eligible for execution of these goals

Every zone in our system is divided into k sectors, $k \in \mathbb{N}$ and $k > 1$. A robot is responsible for cleaning a certain sector. We associate the notion of a *subsubgoal* (or simply *task*) with the process of *cleaning a particular sector*. The task is completed when the sector is cleaned. A new variable *sector -completed* represents the current task status for every sector.

$$sector\_completed \in 1..n(1..k \rightarrow BOOL)$$

The refined model is again built as an instantiation of the *Goal Decomposition Pattern*, where the subsubgoal states are defined as the elements of the variable *sector_completed,* i.e.,

$$gstate_{ij} = sector\_completed(i)(j),\ for\ i \in 1..n,\ j \in 1..k$$

A gluing invariant expresses the relationship between subgoals and tasks:

$$\forall zone \cdot zone \in 1..n \Rightarrow (zone\_completed(zone) = TRUE \Leftrightarrow$$

$$sector\_completed(zone)[1..k] = \{TRUE\})$$

The invariant postulates that any zone is cleaned if and only if its every sector is cleaned. The abstract event CleaningZone is refined by the event CleaningSector. The subsubgoal will be achieved if this section is eventually cleaned:

Machine MRS_Ref2
CleaningSector $\widehat{=}$ refines CleaningZone
status $anticipated$
any $zone, sector, sector\_result$
when
$zone \in 1..n \ \wedge sector \in 1 \ldots k \ \wedge$
$sector\_completed(zone)(sector) = FALSE \ \wedge$
$sector\_result \in BOOL$
then
$sector\_completed(zone) := sector\_completed(zone)$
$\Leftarrow \{sector \mapsto sector\_result\}$
end

Now we have reached the desire level of granularity of our subgoals. In the next refinement step (the machine MRS_Ref3) we are going to augment our model with an abstract representation of agents.

Third refinement. The next refined model of our development is constructed according to the refinement *Agent Modelling Pattern*. As a result, we introduce the abstract set *AGENTS*, and its subset *ELIG* containing the eligible agents for executing the tasks. A new variable *elig* represents the dynamic set of (currently available) eligible agents. Following the proposed pattern, we should also guarantee that there will be at least one eligible agent for cleaning the sector. This property is formulated as an additional invariant: $elig \neq \varnothing$.

Moreover, according to the pattern, we need abstractly introduce agent failures. This is achieved by refining the abstract event CleaningSector by two events SuccessfulCleaningSector and FailedCleaningSector, which respectively model successful and unsuccessful execution of the task by some eligible agent:

Machine MRS_Ref3
SuccessfulCleaningSector $\widehat{=}$ refines CleaningSector
status $convergent$
any $zone, sector, ag$
when
$zone \in 1..n \wedge sector \in 1..k \wedge$
$sector\_completed(zone)(sector) = FALSE \wedge ag \in elig$
then
$sector\_completed(zone) := sector\_completed(zone)$
$\Leftarrow \{sector \mapsto TRUE\}$
end
FailedCleaningSector $\widehat{=}$ refines CleaningSector
status $convergent$
any $zone, sector, ag$
when
$zone \in 1..n \wedge sector \in 1..k \wedge$
$sector\_completed(zone)(sector) = FALSE \wedge$
$ag \in elig \wedge card(elig) > 1$
then
$sector\_completed(zone) := sector\_completed(zone)$
$\Leftarrow \{sector \mapsto FALSE\}$
$elig := elig \setminus \{ag\}$
end

Following the proposed pattern, we add in the event FailedCleaningSector the guard *card*(*elig*) > 1 to restrict possible agent failure in task performance. Let us also note that for multi-robotic systems with many homogeneous agents this constraint is not unreasonable. This assumption allows us to prove the convergence of the goal-reaching events, i.e., to prove that the process of cleaning the territory eventually terminates.

Fourth refinement. Finally, the *Agent Refinement Pattern* for introducing agent types and their statuses is applied to produce the last refined model of our multi-robotic system. In this refinement step we explicitly define the agent types - robots and base stations. We partition our abstract set *AGENTS* by disjointed non-empty subsets *RB* and *BS*, that represent robots and base stations respectively. In this case study robots perform the cleaning task. Hence our abstract set of eligible agents is completely represented by robots: *ELIG* = *RB*. Robots might

be active or failed. We introduce the enumerated set *STATUS*, which in our case has two elements *{active, failed}*.

At previous refinement step we have modelled agent faults while performing their tasks in a very abstract way. Now we will specify them more concretely. We assume that only robots may fail in our multi-robotic system. Their dynamic status is stored in the variable *rbstatus:*

$$rb\_status \in RB \to STATUS.$$

The abstract variable *elig* is now data refined by the concrete set:

$$elig = \{a \mid a \in AGENTS \wedge atype(a) = RB \wedge rb\_status(a) = active\}.$$

The concrete events are also built according to the proposed pattern. For instance, the event FailedCleaningSector can now be specified as follows:

Machine MRS_Ref4
FailedCleaningSector $\widehat{=}$ refines FailedCleaningSector
any $zone, sector, ag$
when
$zone \in 1..n \wedge sector \in 1..k \wedge$
$sector\_completed(zone)(sector) = FALSE \wedge$
$ag \in RB \wedge card(\{a | a \in RB \wedge rb\_status(a) = active\}) > 1$
$rb\_status(ag) = active$
then
$sector\_completed(zone) := sector\_completed(zone)$
$\Leftarrow \{sector \mapsto FALSE\}$
$rb\_status(ag) := failed$
end

An overview of the development of an autonomous multi-robotic system according to the proposed specification and refinement patterns is shown in the Fig. 5.1.

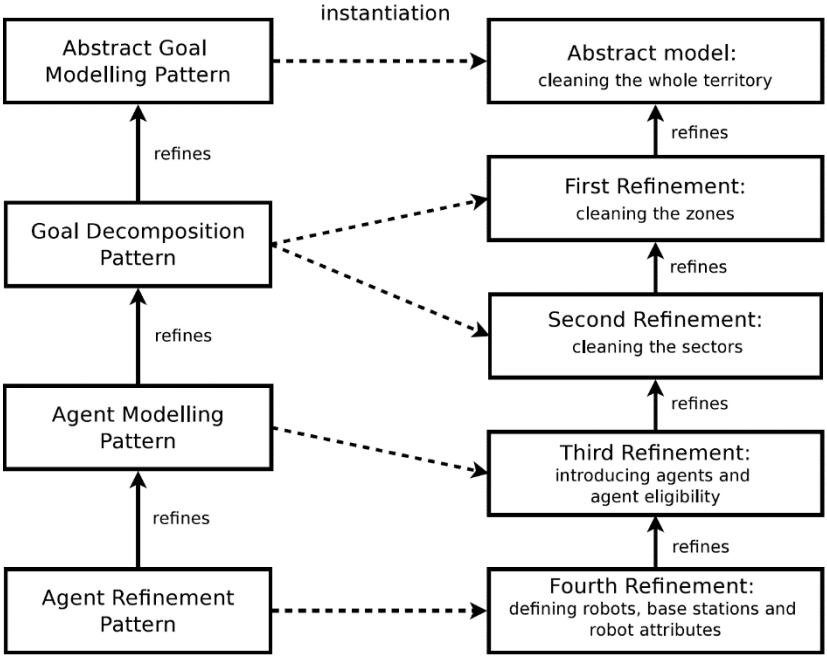

Fig. 5.1. Overview of the development

While modelling the behaviour of a multi-robotic system, we have shown that refinement process allows us also to discover restrictions that we have to impose on system behaviour to guarantee its resilience. In our case, the goal was achievable only if at least one robot remains healthy. Feasibility of such a restriction can be checked probabilistically based on the failure rates of robots.

Tasks for seminar 1.

1. Preparation (determining) of the theme for the work (abstract analytical review, development) and clarifying the tasks.

Topics of work can be formed by students on their own and agreed with the leaders on the basis of the indicative list:

– formal development and quantitative assessment of a resilient multi-robotic system;

– formal reasoning about resilient goal-oriented multi-agent systems;

- integrating event-b modelling and discrete-event simulation to analyse resilience of data stores in the cloud;
- biological immunity and software resilience;
- dynamic software diversity for resilient redundant embedded systems;
- designing a resilient deployment and reconfiguration infrastructure for remotely managed cyber-physical systems.

2. Search of the subject information (library, the Internet) and its preanalysis.

Submission of abstract and presentation in English.

Guidelines and a list of recommended reading to abstracts issued individually.

3. The report plan development and project presentation.

Report plan (and presentation) includes the preparation of the following sections:

- introduction of (motivation, previous works, state-of-art, the main task of the abstract, the structure and characteristics of the content, the work plan);
- a systematic presentation of the basic parts of the report (classification schemes, the characteristic of models, methods, tools, techniques in groups, the choice of indicators and criteria for evaluation, comparative analysis);
- conclusions (achieving statement of the goal, the basic theoretical and practical results, its importance, further work directions);
- references;
- appendix.

4. Report writing. The report shoud has a 15-20 A4 pages (font size 14, half interval, margins 2 cm), including the title page, the content, the main text, references, appendix. Reports prepared by the simple compilation of Internet material without careful structuring, using the incorrect terminology, and without conclusions are not considered.

5. Presentation preparation. The presentation should be designed in PowerPoint and corresponded to the plan of the repotrt (10-15 slides) according to the presentation time - 10 min.

The presentation should include the following slides:

- the title slide (with the theme of the report, the author, date of presentation);

- the content (structure) of the report;
- the motivation of the issues, purpose and tasks of the report on the basis of this analysis;
- slides with highlighted questions according to tasks;
- the conclusions of the report;
- references.

Each slide should contain a footer with the title and author of the report.

Slide content should not be a part of the text of the report, and include keywords, pictures, formulas.

Submission information can be dynamic.

Report defense

Report defense is carried during the seminar, itshould take about 15 minutes and include the actual report with a presentation (10 minutes) and discussion (5 minutes).

Assessment

Assessment takes into account the quality of the report text (form and content), presentations (content and design), the report (structure, content and conclusions), completeness, and correctness of answers.

Advancement questions

1. What frameworks are useful for structuring and specifying complex system requirements?
2. What are the main steps to ensure system resilience in order to guarantee its dependability?
3. What have to do to correct-by-construction system development?
4. What verification is undertaken by abstracting implementation up to requirements level and model-checking satisfiability of goals?
5. What does allow us to ensure that a concrete specification preserves globally observable behavior and properties of abstract specification?
6. What the Event-B framework has been designed for?
7. What does an abstract state machine encapsulate?
8. What a refinements patterns allow perform concerning to te system?

9. What notion is specified using in Event-B?
10. In Event-B, the variables are strongly typed by the constraining predicates called invariants
11. What are the main features in constraining predicates in Event-B?

3.2 Seminar№2. Formal Modelling of Resilient Data Storage in Cloud

The aim and the task of the laboratory work

The aim of this laboratory work is to formalise an industrial approach to implementing resilient cloud data storage.

Task of the work:

- To ensure resilience, F-Secure combined the WAL mechanism with the log replication.
- describe the formally expressed data integrity and consistency properties in three different replication architectures and explicitly identified situations that lead to data loss.
- use modelling approach to facilitate early design exploration and evaluate benefits of different fault tolerance mechanisms in implementing resilience requirements.

Preparation for laboratory work

- to clarify the aims and objectives;
- to study theoretical material given in the description.

Theoretical material

Introduction

Rapid development of digital technology puts a high demand on reliable handling and storage of large volumes of data. It is forecasted that worldwide consumer digital storage needs will grow from 329 exabytes in 2011 to 4.1 zettabytes in 2016 [89]. Often algorithms for data storage in cloud reuse the ones that have been proposed for databases. The transactional model adopted in databases guarantees

ACID properties - Atomicity, Consistency, Isolation and Durability, and as such delivers high resilience guarantees. However, in a pursue of high performance, cloud data storages rarely rely on the transactional model and hence deliver weaker guarantees regarding data integrity. This subsection undertakes a formal study of data integrity and consistency properties that can be guaranteed by several different architectures of cloud data stores.

To achieve a high degree of fault tolerance, let us has combine write-ahead logging (WAL) [81,82] - a widely used mechanism for database error recovery - and massive data replication. As such, this combination gives very high resilience guarantees (usually in the form of eventual consistency). However, these guarantees are different in non-transactional settings typical for cloud. Moreover, data integrity and consistency properties vary in the synchronous, semi-synchronous and asynchronous architectures used for data replication. Therefore, it is useful to rigorously define and compare the properties that can be ensured by different solutions.

To formally model write-ahead logging in replicated data stores the Event-B method and the associated Rodin platform is used. Event-B [86] is a formal framework that is particularly suitable for the development of distributed systems. System development starts from an abstract specification that is transformed into a detailed specification in a number of correctness- preserving refinement steps. Here the synchronous, semi-synchronous and asynchronous replication architectures are separately modelled. Event-B and the Rodin platform [84] allow us to explicitly define the data integrity and consistency properties as model invariants and compare them in all three models. Such approach allows the designers to gain formally grounded insights on properties of cloud data stores and their resilience.

Modelling in Event-B

Event-B is a state-based formal approach that promotes the correct-by-construc- tion development paradigm and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an *abstract state machine* [86]. An abstract state machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic

behaviour of a modelled system. The variables are strongly typed by the constraining predicates that together with other important system properties are defined as model *invariants.* Usually, a machine has an accompanying component, called a *context,* which includes user-defined sets, constants and their properties given as a list of model axioms.

The dynamic behaviour of the system is defined by a collection of atomic *events.* Generally, an event has the following form:

e ≙ any *a* where G_e then R_e end,

where e is the event's name, *a* is the list of local variables, and (the event *guard*) G_e is a predicate over the model state. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment to the system variables. In Event-B, this assignment is semantically defined as the next-state relation R_e.

The event guard defines the conditions under which the event is *enabled,* i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

Event-B employs a top-down refinement-based approach to system development. A development starts from an abstract specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps we gradually reduce nondeterminism and introduce detailed design decisions. In particular, we can add new events, refine old events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement.* In the latter case, we need to define *gluing invariants,* which define the relationship between the abstract and concrete variables. The proof of data refinement is often supported by supplying *witnesses* - the concrete values for the replaced abstract variables. Witnesses are specified in the event clause with.

The consistency of Event-B models, i.e., verification of model well-formedness, invariant preservation as well as correctness of refinement steps, is demonstrated by discharging the relevant proof obligations. The Rodin platform [84] provides an automated support for modelling and verification. In particular, it automatically generates the required proof obligations and attempts to discharge them.

Event-B adopts an event-based modelling style that facilitate a correct-by- construction development of a distributed system. Since cloud data storage is a large-scale distributed system, Event-B is a natural choice for its formal modelling and verification.

Resilient Cloud Data Storage

Essentially, a cloud data storage can be seen as a networked online data storage available for its clients as a cloud service. Data are stored in virtualised data stores (pools) usually hosted by third parties. Physically, the data stores may span across multiple distributed servers. Cloud data storage providers should ensure that their customers can safely and easily store their content and access it from their computers and mobile devices. Therefore, there is a clear demand to achieve both resilience and high performance in handling data.

Write-ahead logging (WAL) is a standard data base technique for ensuring data integrity. The main principle of WAL is to apply the requested changes to data files only after they have been logged, i.e., after the log has been stored in the persistent storage (disk). The WAL mechanism ensures fault-tolerance because, in case of a crash, the system would be able to recover using the log. Moreover, the WAL mechanism helps to optimise performance, since only the log file (rather than all the data changes) should be written to the permanent storage to guarantee that a transaction is (eventually) committed.

The WAL mechanism has been thoroughly studied under the reliable persistent storage assumption, i.e., if the disk containing the log never crashes. However, in the cloud implementing such a highly-reliable data store is rather unfeasible. Therefore, to ensure fault tolerance, F-Secure has proposed a solution that combines WAL with replication. The resulting system - distributed data store (DDS) - consists of a number of nodes distributed across different physical locations. One of the nodes, called *master,* is appointed to serve incoming data requests from DDS clients and report on success or failure of such requests. As a result, for instance, the client may receive an acknowledgment that the data have been successfully stored in the system. The remaining nodes, called *standby nodes,* contain replicas of the stored data.

Each request received by the master is translated into a number of reading and writing commands. These commands are first recorded in the *master log* and then applied to the stored data. After this, an acknowledgement is sent to the client. (In the non-replicated version of WAL widely used in the databases, an acknowledgement to the client is sent already after the request is written in the log). The standby nodes are constantly monitoring and streaming the master log records into their own logs, before applying them to their persistent data in the same way. Essentially, the standby nodes are continually trying to "catch up" with the master. If the master crashes, one of the standby nodes is appointed to be the master in its stead. At this point, the appointed standby effectively becomes the new master and starts serving all data requests.

DDS can implement different models (architectures) of logging. In the asynchronous model, the client request is acknowledged after the master node has performed the required modifications in its persistent storage. The second option - the cascade master-standby - is a semi-synchronous architecture. The client receives an acknowledgement after both the master and its warm standby (called upper standby) has performed the necessary actions. Finally, in the synchronous model, only after all replica nodes have written into their persistent storage, i.e., fully synchronised with the master node, the transaction can be committed. Obviously, such different logging models deliver different resilience guarantees.

In the formal modelling, we aim at formally defining and comparing data integrity and consistency properties that can be ensured by each architecture.

Modelling the Asynchronous Architecture

In the asynchronous model of replication, the standby nodes may stream the master log records only after the required changes have been committed and reported to the client. If the master crashes shortly after committing the required modifications, some changes will not be replicated thus leading to an inconsistent system state. In particular, this might happen because a standby node has not yet received (streamed) all the master log records when the master failed. To minimise such a data loss, the node that has the freshest (and hence the most complete)

copy of the master log is chosen to become the next master. A graphical representation of the system architecture is shown in Fig.6.1.

Abstract specification. The initial model - the machine Replicationl_m0 abstractly describes the behaviour of the master node - receiving and processing of the received requests. The overall model structure is given on Fig.6.2.

The variable *comp, comp* $\subseteq$ *COMP*, represents the dynamic set of active system nodes (data stores), where *COMP* is a set (type) of all available data stores. The variable **master,** such that **master** $\in$ **comp,** represents the master node. The other variables **buffer, inprocess** and **processed** represent the received data requests at different stages of their processing by the master. They are modelled as disjoint sets of the abstract data type **REQUESTS.** In particular, the variable **buffer** stores the requests that have been received by the master and are waiting to be handled. The variable **inprocess** contains the requests that the master node is currently processing, while the variable **processed** keeps the requests that are completed and acknowledged to the client.

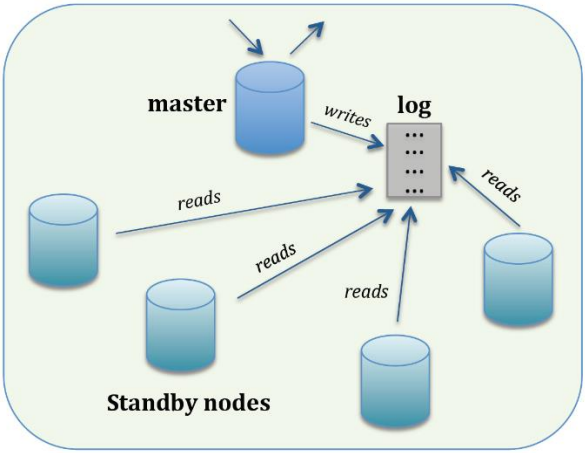

Figure.6.1. A graphical representation of the system architecture

The event **Requestln** specifies arriving of a new request to the master. Processing of the received requests and sending notifications to the client are modelled by the events **Process** and **RequestOut**

respectively. The events update the variables **buffer, inprocess** and **processed** to reflect the progress in request handling.

Machine Replication1_m0
Variables $comp, master, buffer, inprocess, processed$
Invariants $comp \subseteq COMP \wedge master \in comp \wedge buffer \subseteq REQUESTS$ ...
Events

RequestIn $\widehat{=}$ // arriving of a new request
  any r
  where $r \in REQUESTS \wedge r \notin buffer \wedge r \notin inprocess \wedge r \notin processed$
  then $buffer := buffer \cup \{r\}$
  end

Process ... // request processing

RequestOut $\widehat{=}$ // completion of a request
  any r
  where $r \in inprocess$
  then $processed := processed \cup \{r\} \parallel inprocess := inprocess \setminus \{r\}$
  end

ChangeMaster $\widehat{=}$ // changing the master
  any $new\_master, n\_buffer, n\_inprocess, n\_processed$
  where
    $new\_master \in comp \wedge new\_master \neq master \wedge$
    $n\_inprocess \cap n\_buffer = \varnothing \wedge$
    $n\_inprocess \cap n\_processed = \varnothing \wedge$
    $n\_processed \cap n\_buffer = \varnothing \wedge$
    $n\_buffer \cup n\_inprocess \cup n\_processed \subseteq$
    $buffer \cup inprocess \cup processed$
  then
    $master := new\_master \parallel buffer := n\_buffer \parallel$
    $inprocess := n\_inprocess \parallel$
    $processed := n\_processed \parallel comp := comp \setminus \{master\}$
  end

CompActivation ... // activation of a new system component

CompDeactivation ... // deactivation of a system component

Figure. 6.2. Asynchronous model: the abstract model

The event ChangeMaster models a crash of a master and selection of a new master. One of the remaining nodes is non-deterministically chosen to become a new master, while the old master is removed from the set of active nodes. Due to possible data loss, the requests being handled by the new master may be only a subset of those of the failed master. This is reflected by the guard condition:

$n\_buffer \cup n\_inprocess \cup processed \subseteq buffer \cup inprocess \cup processed$

where n_*buffer, n_inprocess,* and *n_processed* are the corresponding data structures of the new master.

Finally, the last two events, CompActivation and CompDeactivation, model a possibility to add new data storage nodes from the cloud and remove some currently active nodes from the system respectively. Only standby nodes can be activated and deactivated in this way.

First Refinement. In the first refinement step (defined by the machine Replication1_ref1), we extend the abstract model by explicitly representing the behaviour of the standby nodes.

To accomplish this, we lift the abstract variables *buffer, inprocess, processed* to become node-dependent functions. In Event-B, this is achieved by data refinement that replaces these variables with the new variables *comp_buffer, comp-inprocess* and *comp-processed.* The following gluing invariants are defined to to prove correctness of data refinement:

$comp\_buffer \in comp \to \mathrm{P}(REQUESTS) \wedge comp\_buffer(master) = buffer \wedge$

$comp\_inprocess \in comp \to \mathrm{P}(REQUESTS) \wedge comp\_inprocess(master) = inprocess \wedge$

$comp\_processed \in comp \to \mathrm{P}(REQUESTS) \wedge comp\_processed(master) = processed$

The overview of the refined model is presented in the Fig. 6.3. The set of model events includes the refined versions of the abstract events (RequestInMst, ProcessMst, RequestOutMst, ChangeMaster, CompActivation, and CompDeactivation) as well as new events describing the behaviour of standby nodes.

We refine the event ChangeMaster to a deterministic procedure of choosing the node with the freshest log as a new master to the failed master. We formulate this condition as a new guard of the event ChangeMaster in the following way:

$\forall c \cdot c \in comp \wedge c \neq new\_master \wedge c \neq master \Rightarrow$

$comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq$

$comp\_buffer(new\_master) \cup comp\_inprocess(new\_master) \cup comp\_processed(new\_master)$

The standby nodes are continuously streaming the master log. Essentially, this means that, as soon as the master node completes the request(s), i.e., performs the required modifications in its persistent storage, the standby nodes start copying the corresponding entries in the

master log. This behaviour is modelled by the new event RequestInStb. Similarly as for the master node, the processing of requests and their completion by the standby nodes are respectively modelled by the events ProcessStb and RequestOutStb.

In our model, we assume that the nodes might become temporary unavailable (i.e., crush and recover). The new variable *failed, failed* $\subseteq$ *comp,* is introduced to store such failed nodes. The new event CompFailure and CompStbRecovery model possible node crashes and recoveries correspondingly.

Machine Replication1_ref1 **refines** Replication1_m0
Variables $comp, master, comp\_buffer,$
$comp\_inprocess, comp\_processed, failed, in\_transit$
Invariants ...
Events
RequestInMst **refines** RequestIn ... // arriving of a new request to the master
ProcessMst **refines** Process ... // request processing by the master
RequestOutMst **refines** RequestOut ... // completion of a request by the master
RequestInStb $\widehat{=}$ ... // reading the master by a standby
any r
where $c \in comp \wedge c \neq master \wedge r \in comp\_processed(master) \wedge$
$r \notin comp\_buffer(c) \wedge r \notin comp\_inprocess(c) \wedge r \notin comp\_processed(c)$
$c \notin failed \wedge master \notin failed$
then $comp\_buffer(c) := comp\_buffer(c) \cup \{r\}$
end
ProcessStb ... // request processing by a standby
RequestOutStb ... // completion of a request by a standby
ChangeMaster **refines** ChangeMaster $\widehat{=}$ // changing the master
any new_master
where $new\_master \in comp \wedge new\_master \neq master \wedge$
$master \in failed \wedge new\_master \notin failed \wedge$
$(\forall c \cdot c \in comp \wedge c \neq new\_master \wedge c \neq master \Rightarrow$
$comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq$
$comp\_buffer(new\_master) \cup comp\_inprocess(new\_master) \cup$
$comp\_processed(new\_master))$
with $n\_buffer = comp\_buffer(new\_master)$
$n\_inprocess = comp\_inprocess(new\_master)$
$n\_processed = comp\_processed(new\_master)$
then $master := new\_master \parallel comp\_buffer := \{master\} \lhd comp\_buffer \parallel$
$comp\_inprocess := \{master\} \lhd comp\_inprocess \parallel$
$comp\_processed := \{master\} \lhd comp\_processed \parallel$
$comp := comp \setminus \{master\} \parallel failed := failed \setminus \{master\} \parallel$
$in\_transit := TRUE$
end
CompActivation ... // activation of a new component into the system
CompDeactivation ... // deactivation of a component from the system
CompFailure ... // modelling a component failure
CompStbRecovery ... // recovery of a standby
TransitionOver ... // all the standby nodes have only the requests already processed by the new master

Figure 6.3. Asynchronous model: the first refinement

Now we are ready to formulate and prove some data consistency properties expressing the relationships between the requests handled by the master and those handled by the standby nodes. Since any standby node is continuously copying the master log, we can say that any standby node is logically "behind" the master node. Mathematically, this means that all the standby requests (no matter what stage of processing they are in) are subset of those of the master node.

Moreover, all the requests that are now handled by a standby node should have been already completed by the master before. We can formulate these two properties as the following system invariants:

$$(\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq comp\_buffer(master) \cup comp\_inprocess(master) \cup comp\_processed(master)), \quad (2)$$

$$(\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq comp\_processed(master)). \quad (3)$$

As it turns out, the last property cannot be proven as an (unconditional) invariant of the system. Indeed, it can be violated right after one of the standby nodes is appointed the new master. A short transitional period may be needed for the new master to "catch up" with some of the standby nodes that got ahead by handling the requests still not committed by the new master. It is easy to show termination of this transitional period, since all such standby nodes are blocked from reading any new requests from the master until the master catches up with them by processing its requests.

We can formally model this transitional stage by introducing the variable *in-transit, inJtransit* $\in$ *BOOL.* The variable obtains the value *TRUE* when a new master is appointed, and reobtains the value *FALSE* (in the new event TransitionOver) when all the remaining standby nodes have the requests already processed by the new master.

Then we can reformulate the property (3) as a system invariant and prove its preservation:

$$in\_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow comp\_buffer \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq comp\_processed(master)). \quad (4)$$

Second Refinement. In the previous refinement step we introduced the standby nodes and their interactions with the master. We also modelled how the received data requests are transferred through the different processing stages on the master and standby sides. The variables *buffer, inprocess* and *processed* were used to store incoming, processing and processed requests. The goal of the second refinement step is explicitly model the WAL mechanism and the resulting inter-dependencies between the master and standby logs/

Mathematically, any log can be represented as a sequence, i.e., as a function of the type

$$any\_\log \in 1..k \rightarrow ELEMENTS$$

where *k* is the index of the last written element.

In our case, we want to store in the node log all the requests - received, being processed, or completed. This can be represented as partitioning of the component log into three separate parts. To achieve that, we introduce three variables *index_written, index_inprocess,* and *index_processed:*

$$index\_written \in comp \rightarrow NAT,\ index\_inprocess \in comp \rightarrow NAT,$$

$$index\_processed \in comp \rightarrow NAT,$$

such that

$$\forall c \cdot c \in comp \Rightarrow index\_inprocess(c) \leq index\_written(c),$$

$$\forall c \cdot c \in comp \Rightarrow index\_processed(c) \leq index\_inprocess(c).$$

For any component c, *index_written(c)* defines the index of the last written log entry, *index_inprocess(c)* - the index of the last request being processed, and *index_processed(c)* - the index of the last completed request. Graphically, this can be represented as shown in Fig.6.4.

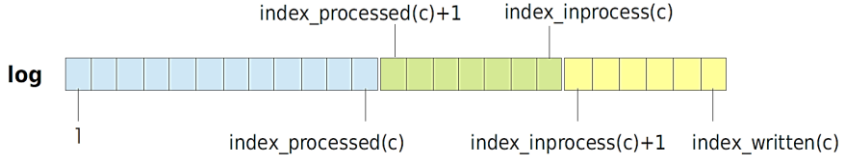

Figure. 6.4. The log partition

Then the logs of all the components can be defined as the following function:

$\log \in comp \to (NAT \to REQUESTS)$,

such that

$\forall c \in comp \cdot dom(\log(c)) = 1..index\_written(c)$,

where dom is the functional domain operator.

The function *log* is introduced to replace (data refine) the abstract variables *comp_buffer*, *comp-inprocess*, and *comp-processed*. To prove correctness of such data refinement, the following gluing invariants are added:

$$\forall c \cdot c \in comp \Rightarrow log(c)[index\_inprocess(c) + 1 \,..\, index\_written(c)] = comp\_buffer(c),$$

$$\forall c \cdot c \in comp \Rightarrow log(c)[index\_processed(c) + 1 \,..\, index\_inprocess(c)] = comp\_inprocess(c),$$

$$\forall c \cdot c \in comp \Rightarrow log(c)[1 \,..\, index\_processed(c)] = comp\_processed(c),$$

where R[S] denotes relational image of *R* with respect to the given set *S*.

An introduction of the sequential representation of the component log allows us to refine some proven invariants as well as prove some new ones. For instance, the invariant property (4) now can be reformulated in terms of new variables

$$in\_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow log(c)[1 \,..\, index\_written(c)] \subseteq log(master)[1 \,..\, index\_processed(master)]). \quad (5)$$

The formulated data refinement also affects all the events where the abstract variables were used. For instance, the event RequestOutMst (see Fig. 6.5) now specifies completion of master request processing by recording this in the node log, i.e., by increasing *index_processed(master).*

We can refine the procedure of choosing a new master by reformulating the guard condition (1) of the event ChangeMaster as follows:

$$\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq new\_master \Rightarrow index\_written(c) \leq index\_written(new\_master). \quad (6)$$

Here we check that the new candidate for the master has the largest *index_written,* i.e., the freshest log copy. The other events are refined in a similar way. The overview of the refined model is presented in Fig. 5.6. Moreover, we can explicitly formulate and prove the log data integrity properties as model invariants:

$$\forall c, i \cdot c \in comp \wedge i \in 1 \,..\, index\_written(c) \Rightarrow log(c)(i) = log(master)(i),$$

$$\forall c1, c2, i \cdot c1 \in comp \wedge c2 \in comp \wedge i \in 1 \,..\, index\_written(c1) \wedge i \in 1 \,..\, index\_written(c2) \Rightarrow log(c1)(i) = log(c2)(i). \quad (7)$$

Machine Replication1_ref2 refines Replication1_ref1
Variables $comp, master, comp\_buffer,$
$comp\_inprocess, comp\_processed, failed, in\_transit$

Invariants...
Events
RequestInMst $\widehat{=}$ refines RequestInMst ...
// arriving of a new request to the master
ProcessMst refines ProcessMst ...
// request processing by the master
RequestOutMst $\widehat{=}$ refines RequestOutMst ...
// completion of a request by the master
when $index\_processed(master) \neq index\_inprocess(master) \wedge master \notin failed$
with $r = log(master)(index\_processed(master) + 1)$
then $index\_processed(master) := index\_processed(master) + 1$
end
RequestInStb $\widehat{=}$ refines RequestInStb ...
// reading the master by a standby
any c
where $c \in comp \wedge c \neq master \wedge c \notin failed \wedge master \notin failed \wedge index\_written(c) < index\_processed(master)$
with $r = log(master)(index\_written(c) + 1)$
then $log(c) := log(c) \cup \{index\_written(c) + 1 \mapsto log(master)(index\_written(c) + 1)\} \parallel index\_written(c) := index\_written(c) + 1$
end
ProcessStb $\widehat{=}$ refines ProcessStb ...
// request processing by a standby
RequestOutStb $\widehat{=}$ refines RequestOutStb ...
// completion of a request by a standby
ChangeMaster $\widehat{=}$ refines ChangeMaster ...
// changing the master
...

Figure 6.5. Asynchronous model: the second refinement

These properties state that the corresponding log elements of any two storage (master or standby) nodes are always the same. In other words, all logs are consistent with respect to the log records of the master node.

The Cascade Master-Standby and Synchronous Architectures

An alternative, semi-syncronous replication model is the *cascade master-standby*. Besides the *master* node that serves incoming data base requests, we single out another functional node - *upper standby*. The upper standby node starts streaming the master log as soon as the master records the requests in its log. Moreover, the master node waits until the upper standby reads its processed records and, only after that, commits the changes and reports to the client.

In its turn, the other standby nodes are constantly monitoring and streaming the upper standby log records into their own logs. Essentially, the standby nodes are continually trying to catch up with the upper standby.

If the master node goes down, the upper standby node is automatically appointed to be the master in its stead. Moreover, the next candidate for the new upper standby node becomes the node that is closest (with respect to the copied log file) to the current upper standby.

Let us note that this proposed cascade replication mode allows to decrease the possibility of loss of the committed changes if the master node fails. Indeed, at that point, when the master node fails, the upper standby node had already recorded all the changes that were committed and reported to the client by master before. Therefore, such an architectural solution increases the system resilience. A possibility of data loss leading to an inconsistent system state is still present. However, for this to happen, the master node and the upper standby node should both fail in a very short time period. A graphical representation of the system architecture is shown in Fig.6.6.

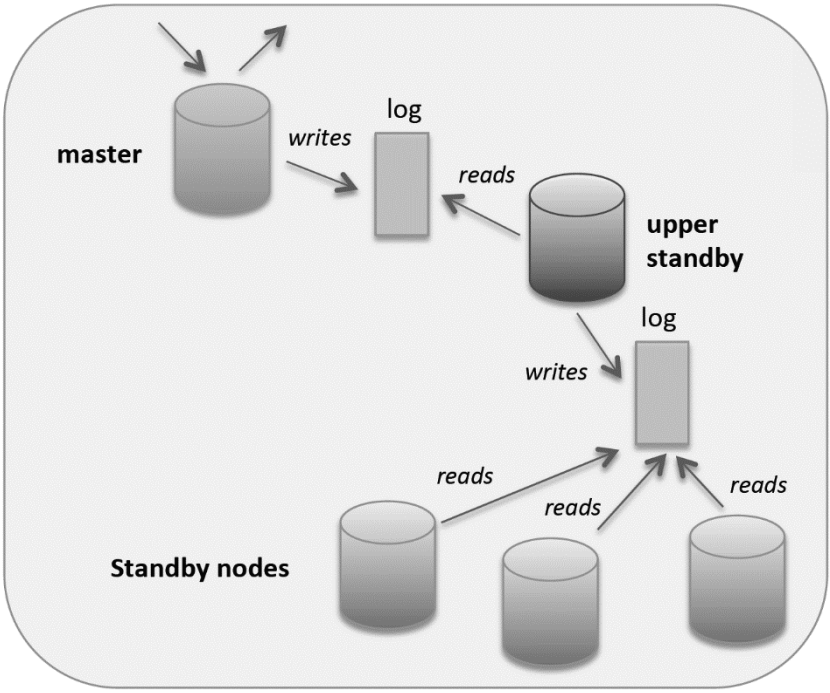

Fig. 6.6. Cascade system architecture

The formal development of the proposed replication model consists of an initial specification and its two refinements. The initial model abstractly describes the system behaviour focusing on the master and the upper standby nodes. The first refinement step introduces the remaining standby nodes and their interoperation with the upper standby, while the second refinement explicitly models the sequential logging mechanism and the interdependencies between the master, the upper standby and others standby logs. Let us note that the development is similar to that of the asynchronous model. Due to the lack of space we will only highlight the most significant differences between them.

Abstract specefication. In the initial model defined by the machine Replication2_m0 we focus on the master and upper standby components and their interoperation. The overall model structure is given on Fig.6.7.

In addition to the master node,we single out one more node to serve as an upper standby node. We model this by inrodusing the variable ups_stanbdy, such that ups_stanbdy $\in$ comp and ups_stanbdy $\neq$ master

The variables *m_buffer, m_inprocess, m_processecl* represent the received requests at different stages of their processing by the master. Similarly, the variables *ups_buffer, ups_inprocess, ups_processed* are introduced to model the respective data structures for the upper standby. The events *RequestlnMst, ProcessMst, RequestOutMst* and *RequestlnUps, ProcessUps, RequestOutUps* specify the corresponding request stages for the master and upper standby nodes.

The master node can not commit the changes until the upper standby reads them. We model this requirement by adding the following guard condition in the event *RequestOutMst:*

$$r \in ups\_buffer \cup ups\_inprocess \cup ups\_processed.$$

The process of changing of the master node by the upper standby is modelled by the event *ChangeMaster.* The event also specifies the selection procedure of a new upper standby. Due to possible data loss, the requests being handled by the new upper standby may be only a subset of those of the current upper standby:

$$n\_buffer \cup n\_inprocess \cup n\_processed \subseteq$$
$$ups\_buffer \cup ups\_inprocess \cup ups\_processed.$$

Machine Replication2_m0
Variables $comp, master, ups\_standby,$
$m\_buffer, m\_inprocess, m\_processed, m\_buffer, \ldots$
Invariants $comp \subseteq COMP \wedge master \in comp \wedge$
$ups\_standby \in comp \ldots$

Events ...
RequestInMst ... // arriving of a new request to the master node
ProcessMst ... // request processing
RequestOutMst $\widehat{=}$ // completion of a request by the master
any r
where $r \in m\_inprocess \wedge$
$(r \in ups\_buffer \cup ups\_inprocess \cup ups\_processed)$
then $m\_processed := m\_processed \cup \{r\} \parallel$
$m\_inprocess := m\_inprocess \setminus \{r\}$ **end**

ProcessUps ... // request processing by the upper standby
ChangeMaster $\widehat{=}$ // changing the master
any $new\_ups\_standby, n\_buffer, n\_inprocess, n\_processed$
where
$new\_ups\_standby \in comp \wedge$
$new\_ups\_standby \neq ups\_standby \wedge$
$new\_ups\_standby \neq master \wedge \ldots$
$(n\_buffer \cup n\_inprocess \cup n\_processed \subseteq$
$ups\_buffer \cup ups\_inprocess \cup ups\_processed)$
then
$master := ups\_standby \parallel ups\_standby := new\_ups\_standby \parallel$
$m\_buffer := ups\_buffer \parallel m\_inprocess := ups\_inprocess \parallel$
$m\_processed := ups\_processed \parallel comp := comp \setminus \{master\} \ldots$
end
ChangeUpsStb ... // changing the upper standby node
CompActivation ... // activation of a new system component
CompDeactivation ... // deactivation of a system component

Fig. 6.7. Cascade architecture: abstract model

Moreover, a similar event, ChangeUpsStb, models the selection of a new upper standby in the case when the current one fails.

First Refinement. In the first refinement step we extend the abstract model by explicitly introducing the behaviour of the remaining

standby nodes. Similarly as for the asynchronous model, we data refine the abstract variables *m_buffer, m_inprocess, m_processed* and *ups_buffer, ups_inprocess, ups_processed* by the new functional variables *comp_buffer, comp_inprocess* and *comp_processed.*

In addition, a number of the new events are added to describe the behaviour of standby nodes, node failures and recovery (RequestInStb, ProcessStb, RequestOutStb, CompFailure, CompStbRecovery).

$$(\forall c \cdot c \in comp \wedge c \neq master \ \wedge c \neq ups\_standby \Rightarrow comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq comp\_buffer(ups\_standby) \cup comp\_inprocess(ups\_standby) \cup comp\_processed(ups\_standby)), \quad (8)$$

$$comp\_buffer(ups\_standby) \cup comp\_inprocess(ups\_standby) \cup comp\_processed(ups\_standby) \subseteq comp\_buffer(master) \cup comp\_inprocess(master) \cup comp\_processed(master), \quad (9)$$

As for the asynchronous model, we can formulate and prove data consistency properties between the involved components. The property (2) (stating that a standby node is always behind the master in terms of handled requests) corresponds to two properties for the cascade replication mode: the first one stating this property between any standby and the upper standby, while the second one stating the same property between the upper standby and master nodes.

The property (4) for the asynchronous mode expresses the relationships between the processed requests of the master node and read requests of the standby nodes. This property again corresponds to two properties for the cascade mode: one between the upper standby and remaining standbys, and the other one between the master and upper standby nodes. In both cases, the properties may be violated for a short period (indicated by *in_transit = TRUE)* right after a new upper standby node is chosen to replace a failed one:

$$in\_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq ups\_standby \Rightarrow (comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c) \subseteq comp\_processed(ups\_standby)), \quad (10)$$

$$in\_transit = FALSE \Rightarrow (comp\_processed(master) \subseteq comp\_buffer(ups\_standby) \cup comp\_inprocess(ups\_standby) \cup comp\_processed(ups\_standby)). \quad (11)$$

Note how the requirement that the master cannot commit a request before it is read by the upper standby reverses the inclusion relationship in the (**11**).

Second Refinement. The goal of the second refinement step is explicitly model the write-ahead logging mechanism and the resulting interdependencies between the master, upper standby and other standby logs.

$$\forall c \cdot c \in comp \Rightarrow log(c)[index\_inprocess(c) + 1 \,..\, index\_written(c)] = comp\_buffer(c),$$

$$\forall c \cdot c \in comp \Rightarrow log(c)[index\_processed(c) + 1 \,..\, index\_inprocess(c)] = comp\_inprocess(c),$$

$$\forall c \cdot c \in comp \Rightarrow log(c)[1 \,..\, index\_processed(c)] = comp\_processed(c).$$

We data refine the abstract variables *comp_buffer*, *comp_inprocess,* and *comp_processed* by the introduced function *log.* The following gluing invariants allow us to prove correctness of such a data refinement:

$$in\_transit = FALSE \Rightarrow$$
$$(\forall c \cdot c \in comp \wedge c \neq master \wedge c \neq ups\_standby \Rightarrow$$
$$log(c)[1 \,..\, index\_written(c)] \subseteq$$
$$log(ups\_standby)[1 \,..\, index\_processed(ups\_standby)]). \quad (12)$$
$$in\_transit = FALSE \Rightarrow log(master)[1 \,..\, index\_processed(master)]$$
$$\subseteq log(ups\_standby)[1 \,..\, index\_written(ups\_standby)]. \quad (13)$$

Introducing the sequential representation of the component log allows us to reformulate some proven invariants as well as prove some new ones. For instance, the invariant properties (**10**) and (**11**) now can be reformulated in terms of the new variables as follows:

Finally, the log data integrity properties (in the exact form as in (7)) are formulated and proved for this replication mode as well.

Synchronous Architecture. The last development formalises the *synchronous replication architecture*, which can be considered as a combination of both asynchronous and cascade models. The essential differences of this model are following. The standby nodes start streaming the master log records as soon as master records the commands in its log. Moreover, the master node waits until *all the standby nodes* read processed records from its log and, only after that, commits the corresponding changes and reports to the client. If the master goes down, one of the standby nodes is appointed to be the master in its stead. Essentially, it is a generalisation of the cascade model where all the standby nodes play the role of upper standby.

This architecture allows to avoid a possibility of loss of the committed changes if the master fails. Indeed, at that point, all the standby nodes have already recorded all the changes that were committed and reported to the client by master. On the other hand, the necessity for the master to synchronise in such a way with all the standbys may negatively affect the performance of this model.

Developing the formal model of this architecture, we essentially repeat the refinement steps of the asynchronous model. In particular, the initial model is the same as the abstract model presented on Fig.6.2. In the first refinement step, in the RequestOutMst event modelling the commitment of the changes by master, we have to impose an additional restriction for this behaviour. Namely, the master node can not commit the changes until all the standby nodes have read them. We model this requirement by adding the following guard condition to the event:

$$\forall c \cdot c \in comp \wedge c \neq master \Rightarrow$$
$$r \in comp\_buffer(c) \cup comp\_inprocess(c) \cup comp\_processed(c),$$

where we check that the request r has already been recorded by all the standby nodes. Moreover, in the eventRequestInStb, we relax its guard by allowing to copy the master log as soon as the master records requests in its log.

Similarly as for the first two models, we formulate and prove log data consistency properties. Specifically, the property (2), stating that the standby nodes are continuously trying to catch up with the master in terms of handled requests, can be proved for this architecture as well. Moreover, since the master can not commit the changes until the all standbys have read the corresponding log records, it means that all the requests committed by the master have been previously read by all standbys. We can formulate this property as follows:

$$in\_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow$$
$$comp\_processed(master) \subseteq comp\_buffer(c) \cup$$
$$comp\_inprocess(c) \cup$$
$$comp\_processed(c)). \quad (14)$$

Note that, once again, this property can be violated right after a new master is appointed and thus a transitional period is needed. This property is very similar to that of (**11**) (for the cascade architecture) and is inverse, with respect to the inclusion relation, to that of (4) (for the asynchronous architecture).

As in the previous two developments, in the second refinement step we introduce component logs as sequences. In terms of the new variables, the (14) property can be then reformulated as follows:

$$in\_transit = FALSE \Rightarrow (\forall c \cdot c \in comp \wedge c \neq master \Rightarrow$$
$$log(master)[1 \,..\, index\_processed(master)] \subseteq$$
$$log(c)[1 \,..\, index\_written(c)]). \quad (15)$$

Finally, the log data integrity properties (7), stating that the corresponding log elements of any two storage components are always the same, are proved for this model as well. The full formal developments can be found in [88].

Proof Statistics To verify correctness of the presented formal developments, we have discharged around 400 proof obligations for the first model, more than 750 proof obligations for the second model, and around 400 for the third model. In total, around 90% of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment. The proof statistics in terms of generated proof obligations for the presented Event B developments is shown in the Table 6.1. The numbers represent the total number of proof obligations and the percentage of manual effort for each model in each refinement step. The whole development and proving effort has taken around one person-month.

Table 6.1. The proof statistics

	Asynchronous model			Cascade model			Synchronous model		
step	Total	Manual	Manual %	Total	Manual	Manual %	Total	Manual	Manual %
m0	18	0	0	53	0	0	18	0	0
ref1	145	0	0	257	1	0.3	146	0	0
ref2	193	42	21.7	442	75	16.9	232	42	18.1
Overall	356	42	11.7	752	76	10.1	396	42	10.6

Tasks for seminar 2.

1. Preparation (determining) of the theme for the work (abstract analytical review, development) and clarifying the tasks.

Topics of work can be formed by students on their own and agreed with the leaders on the basis of the indicative list:

– empirical assessment of resilience;

– quantitative verification of system safety in event-B;

- architecting resilient computing systems;

- predictability and evolution in resilient systems;

- modelling resilience of data processing capabilities of CPS;

- safety lifecycle development process modeling for embedded systems.

2. Search of the subject information (library, the Internet) and its preanalysis.

Submission of abstract and presentation in English.

Guidelines and a list of recommended reading to abstracts issued individually.

3. The report plan development and project presentation.

Report plan (and presentation) includes the preparation of the following sections:

- introduction of (motivation, previous works, state-of-art, the main task of the abstract, the structure and characteristics of the content, the work plan);
- a systematic presentation of the basic parts of the report (classification schemes, the characteristic of models, methods, tools, techniques in groups, the choice of indicators and criteria for evaluation, comparative analysis);
- conclusions (achieving statement of the goal, the basic theoretical and practical results, its importance, further work directions);
- references;
- appendix.

4. Report writing. The report shoud has a 15-20 A4 pages (font size 14, half interval, margins 2 cm), including the title page, the content, the main text, references, appendix. Reports prepared by the simple compilation of Internet material without careful structuring, using the incorrect terminology, and without conclusions are not considered.

5. Presentation preparation. The presentation should be designed in PowerPoint and corresponded to the plan of the repotrt (10-15 slides) according to the presentation time - 10 min.

The presentation should include the following slides:

- the title slide (with the theme of the report, the author, date of presentation);
- the content (structure) of the report;
- the motivation of the issues, purpose and tasks of the report on the basis of this analysis;
- slides with highlighted questions according to tasks;
- the conclusions of the report;
- references.

Each slide should contain a footer with the title and author of the report.

Slide content should not be a part of the text of the report, and include keywords, pictures, formulas.

Submission information can be dynamic.

Report defense

Report defense is carried during the seminar, itshould take about 15 minutes and include the actual report with a presentation (10 minutes) and discussion (5 minutes).

Assessment

Assessment takes into account the quality of the report text (form and content), presentations (content and design), the report (structure, content and conclusions), completeness, and correctness of answers.

Advancement questions

1. What does the F-Secure do to ensure resilience?
2. What are the main steps to describe the formally expressed data integrity and consistency properties in three different replication architectures and explicitly identified situations that lead to data loss?
3. 2.Q. ?
4. What have we do to facilitate early design exploration and evaluate benefits of different fault tolerance mechanisms in implementing resilience requirements?
5. What can we use to write-ahead logging in replicated data stores?
6. What are the main phases to construct the detailed specification in a number of correctness-preserving refinement steps?
7. What does allow us to explicitly define the data integrity and consistency properties as model invariants and compare them in all three models?
8. What does promotes Event-B?
9. What does abstract state machine describe?
10. What does abstract state machine include?

REFERENCES

References for preparation for laboratory work № 1

1. Riham Hassan Abdel-Moneim Mansour Formal Analysis and Design for Engineering Security / Riham Hassan Abdel-Moneim Mansour // PhD thesis. - Blacksburg, Virginia, 2009.
2. Cheng B. Using Security Patterns to Model and Analyze Security Requirements: Technical Report MSU-CSE-03-18 / Cheng B. - Michigan : Michigan State University, 2003.
3. Schumacher M. Security Patterns - Integrating Security and Systems Engineering / M. Schumacher. : John Wiley & Sons, 2005.
4. Fontaine P.J. Goal-Oriented Elaboration of Security Requirements / P.J. Fontaine // PhD Thesis. - Louvain : University of Louvain, 2001.
5. Letier E. Reasoning About Agents in Goal-Oriented Requirements Engineering / Letier E. // PhD Thesis Louvain : Universite Catholique de Louvain, Louvain-la-Neuve, 2001.
6. Amoroso E. J. Fundamentals of Computer Security / Amoroso E.J. - Prentice-Hall, 1994.
7. van Lamsweerde Handling Obstacles in Goal-Oriented Requirements Engineering / A. van Lamsweerde and E. Letier // IEEE Transactions on Software Engineering. Special Issue on Exception Handling, Vol. 26 No. 10, 2000.
8. Schumacher M. Security Patterns - Integrating Security and Systems Engineering / M. Schumacher. - John Wiley & Sons, 2005.
9. Schneider Steve The b-method" / Steve Schneider. - PALGRAVE, 2001.
10. Edmund Wing J. Formal Methods: State of the Art and Future Directions / Edmund Wing J. . - ACM Computing Surveys, Vol. 28, No. 4, 1996.
11. Lehman M. M. On understanding Laws, evolution and conversation in the large program lifecycle / M. M. Lehman // Journal of Software & Systems, vol. 1, 1980.

References for preparation for laboratory work № 2

12. Constance L. Heitmeyer Applying Formal Methods to a Certifiably Secure Software System / Constance L. Heitmeyer, Myla M. Archer, Elizabeth I. Leonard and John D. McLean // IEEE Transactions on Software Engineering. – 2008. - №34 – C.82-98.
13. J. Rushby "Design and Verification of Secure Systems" / J. Rushby // Proc. Eighth ACM Symp. Operating System Principles, 1981.
14. B. Lampson Protection / B. Lampson // Proc. Fifth Princeton Conf. Information Sciences and Systems, 1991.
15. Shankar N. PVS Prover Guide Version 2.4 : technical report / N. Shankar, S. Owre, J.M. Rushby, D.W.J. Stringer-Calvert. - Computer Science Laboratory, 2001.
16. Anderson J. "Computer Security Technology Planning Study" : Technical Report ESD-TR-73-51 / Anderson J. - Hanscom AFB : ESD/AFSC, 1972.
17. Common Criteria for Information Technology Security Evaluation, Parts 1-3: Technical Reports CCIMB-2004-01-001 through CCIMB-2004-01-003, Version 2.2, Revision 256, Jan. 2004.
18. Archer M. Proving Invariants of I/O Automata with TAME / M. Archer, C.L. Heitmeyer, and E. Riccobene // Automated Software Eng. – 2002. - vol. 9, no. 3. - p. 201-232.
19. J. Owre. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS / J. Owre, N. Rushby, D. Shankar, F. von Henke // IEEE Trans. Software Eng. – 1995. - №21. - p. 107-125.
20. Xcode Version 8 [Electronic resource]: Xcode 8. – Access mode : http://developer.apple.com/tools/xcode/index.html. - Name from the screen.
21. M. Abadi. The Existence of Refinement Mappings / M. Abadi, L. Lamport // Theoretical Computer Science. - 1991. №2, vol 21. - p. 253-284.
22. McMillan K.L. Verification of Infinite State Systems by Compositional Model Checking. / K.L. McMillan // Proc. 10th IFIP WG 10.5 Advanced Research Working Conf. Correct Hardware Design and Verification Methods. - 1999.

23. E. Clarke. Counterexample-Guided Abstraction Refinement. /E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith // Proc. 12th Int'l Conf. Computer-Aided Verification. - 2000.
24. B. Alpern. Defining Liveness / B. Alpern and F.B. Schneider // Information Processing Letters. -1985. - №4, vol 21. - p.181-185.
25. L. Lamport. Proving the Correctness of Multiprocess Programs. / L. Lamport. Proving // IEEE Trans. Software Eng. -1977. №2, vol 3. - p.125-143.
26. C.L. Heitmeyer. Tools for Constructing Requirements Specifications: The SCR Toolset at the Age of Ten. / C.L. Heitmeyer, M. Archer, R. Bharadwaj, and R.D. Jeffords // Computer Systems: Science and Eng. - 2005. - №1, vol 20.

References for preparation for laboratory work № 3

27. Bruno Blanchet Using Horn Clauses for Analyzing Security Protocols / Blanchet Bruno // Cryptology and Information Security Series. Models and Techniques for Analyzing Security Protocols. – 2011.- vol 5. – p. 86-111.
28. N. Durgin. Multiset rewriting and the complexity of bounded security protocols. / N. Durgin, P. Lincoln, J. C. Mitchell // Journal of Computer Security. -2004. -№2, Том 12. -С.247-311.
29. L. C. Paulson. The inductive approach to verifying cryptographic protocols. / L. C. Paulson // Journal of Computer Security. - 1998. - №1-2, vol. 6. -С.85-128.
30. C. Weidenbach. Towards an automatic analysis of security protocols in first-order logic. / C. Weidenbach // 16th International Conference on Automated Deduction (CADE-16). - 1999. – vol. 1632. - p.314-328.
31. D. Bolignano. Towards a mechanization of cryptographic protocol verification / D. Bolignano // 9th International Conference on Computer Aided Verification (CAV'97). - 1997. – vol. 1254. – p..131-142.
32. D. Monniaux. Abstracting cryptographic protocols with tree automata. / D. Monniaux // Science of Computer Programming. - 2003. – vol. 2-3, №47. - p.177-202.

33. J. Goubault-Larrecq. A method for automatic cryptographic protocol verification / J. Goubault-Larrecq // Fifth International Workshop on Formal Methods for Parallel Programming: Theory and Applications. - 2000. – vol. 1800. - p.977-984.
34. T. Genet. Rewriting for cryptographic protocol verification. / T. Genet and F. Klay // 17th International Conference on Automated Deduction. - 2000. – vol. 1831. – p.271-290.
35. Y. Boichut. Validation of prouvé protocols using the automatic tool TA4SP. / Y. Boichut, N. Kosmatov, L. Vigneron // Proceedings of the Third Taiwanese-French Conference on Information Technology. - 2006. - p.467-480.
36. C. Bodei Security Issues in Process Calculi / C. Bodei // PhD thesis, 2000.
37. Bodei C. Control flow analysis for the calculus / Bodei C.,Degano P.,Nielson F.,Nielson H. R.In // International Conference on Concurrency Theory. – Springer. – vol. 1466. - 1998 – p. 84-98.
38. Bodei C. Static validation of security protocols / Bodei C., Buchholtz M., Degano P., Nielson F., Nielson H. R. // Journal of Computer Security. – 2005. – 13(3) – p. 347-390.
39. Bozga L. Pattern-based abstraction for verifying secrecy in protocols / Bozga L., Lakhnech Y., Périn M. // International Journal on Software Tools for Technology Transfer. – 2006 – 8(1) – p. 57-76.
40. Backes M. Causality-based abstraction of multiplicity in security protocols / Backes M., Cortesi A., Maffei M. // 20th IEEE Computer Security Foundations Symposium. – 2007 – p. 355-369.
41. Millen K. The Interrogator: Protocol security analysis / Millen K., Clark S.C., Freedman S.B. // IEEE Transactions on Software Engineering. – 1987 – SE-13(2) – p. 274-288.
42. Meadows C.A. The NRL protocol analyzer: An overview / Meadows C.A. // Journal of Logic Programming. – 1996 – vo.26(2) – p. 113-131.
43. Escobar S. A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties / Escobar S., Meadows C., Meseguer J. // Theoretical Computer Science. – 2006 – vol. 367 – p. 162-202.

44. Abadi M. Secrecy types for asymmetric communication / Abadi M., Blanchet B. // Foundations of Software Science and Computation Structures – 2001 – vol. 2030 – p. 25-41.
45. Blanchet B. Automatic verification of correspondences for security protocols / Blanchet B. // Journal of Computer Security. – 2009 – vol. 17(4) – p. 363-434.
46. Filé G. Expressive power of definite clauses for verifying authenticity / Filé G., Vigo R. // 22nd IEEE Computer Security Foundations Symposium. – 2009. - p 251-265.
47. Dolev D. On the security of public key protocols / Dolev D., Yao A.C. // IEEE Transactions on Information Theory. – 1983 – vol. 29(12) – p. 198-208.
48. Abadi M. Mobile values, new names, and secure communication / Abadi M., Fournet C. // 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. – 2001 – p. 104-115.
49. Denker G. Protocol specification and analysis in Maude / Denker G., Meseguer J., Talcott C. // Workshop on Formal Methods and Security Protocols – 1998.
50. Abadi M. Analyzing security protocols with secrecy types and logic programs / Abadi M., Blanchet B. // Journal of the ACM. –2005 – vol. 52(1) – p. 102-146.
51. Blanchet B. Security protocols: From linear to classical logic by abstract interpretation / B.Blanchet // Information Processing Letters. – 2005 – 95(5) – p. 473-479.
52. Bachmair L. Resolution theorem proving / Bachmair L., Ganzinger H. // Handbook of Automated Reasoning. – 2001 – vol .1 – p. 19-100.
53. Blanchet B. Verification of cryptographic protocols / Blanchet B., Podelski A. // Tagging enforces termination. Theoretical Computer Science. – 2005 – vol. 333 – p. 67-100.
54. Abadi M. Mobile values, new names, and secure communication / Abadi M., Fournet C. // 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. – 2001 – p. 104-15.
55. Küsters R. Reducing protocol analysis with XOR to the XOR-free case in the Horn theory based approach / R.Küsters,

T.Truderung // Proceedings of the 15th ACM conference on Computer and communications security. - 2008. – p. 129–138.
56. Küsters R. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation / R.Küsters, T.Truderung // 22nd IEEE Computer Security Foundations Symposium. - 2009. – p. 157–171.
57. Blanchet B. Automatic proof of strong secrecy for security protocols / B.Blanchet // IEEE Symposium on Security and Privacy. - 2004. – p. 86–100.
58. Blanchet B. Automated verification of selected equivalences for security protocols / B.Blanchet, M.Abadi, C.Fournet // Journal of Logic and Algebraic Programming. - 2008. - p.3 –51.
59. Allamigeon X. Reconstruction of attacks against cryptographic protocols / X.Allamigeon, B.Blanchet // 18th IEEE Computer Security Foundations Workshop. - 2005. – p. 140–154.

References for preparation for laboratory work № 4

60. Wihem Arsac Validating Security Protocols under the General Attacker / Wihem Arsac, Giampaolo Bella, Xavier Chantry, Luca Compagna // Lecture Notes in Computer Science. - Springer. – vol. 5511. – p. 34-51.
61. Abadi M. A calculus for cryptographic protocols: the spi calculus / M.Abadi, A.Gordon // Information and Computation 148(1). - 1999. – p.1–70.
62. Ryan P.Y.A. Modelling and Analysis of Security Protocols / P.Y.A.Ryan, S.Schneider, M.Goldsmith, G.Lowe, A.W.Roscoe // AW. - 2001.
63. F´abrega F.J.T. Strand spaces: Proving security protocols correct. / F.J.T.F´abrega, J.C.Herzog, J.D.Guttman // Journal of Computer Security 7. - 1999. – p. 191–230.
64. Caleiro C. Relating strand spaces and distributed temporal logic for security protocol analysis / C.Caleiro, L.Vigan`o, D.Basin // Logic Journal of the IGPL 13(6). - 2005. - pages 637–663.
65. Bella G. Formal Correctness of Security Protocols // Information Security and Cryptography. Springer, Heidelberg. - 2007.

66. Paulson L.C. The inductive approach to verifying cryptographic protocols // Journal of Computer Security. – vol. 6. - 1998. – p.85–128.
67. Blanchet B. Automatic verification of cryptographic protocols: a logic programming approach // Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, Uppsala, Sweden. - 2003. - p. 1–3.
68. Backes M. Relating symbolic and cryptographic secrecy / M.Backes, B.Pfitzmann // RelatingIn: IEEE Symposium on Security and Privacy. - 2005.
69. Bella G. Retaliation: Can we live with flaws? / G.Bella, S.Bistarelli, F.Massacci // Essaidi, M., Thomas, J. (eds.) Proc. of the Nato Advanced Research Workshop on Information Security Assurance and Security. Nato through Science. - IOS Press. – 2006. - vol. 6. – p. 3–14.
70. Bella G. The Rational Attacker [electronic resource] / G. Bella G. // Home page. - Access mode : http://www.dmi.unict.it/~giamp/Seminars/rationalattackerSAP08.pdf – Name from the screen.
71. Bella G. Confidentiality levels and deliberate/indeliberate protocol attacks / G.Bella, S.Bistarelli // Security Protocols. – Heidelberg : Springer. - 2004. - vol. 2845. – p. 104–119.
72. Aiyer A.S. Bar fault tolerance for cooperative services / A.S.Aiyer, L.Alvisi, A.Clement, M.Dahlin, J.P.Martin, C.Porth // ACM SIGOPS Operating Systems Review. – vol. 39(5). - 2005. – p. 45–58.
73. Butty´an L. A formal model of rational exchange and its application to the analysis of syverson's protocol / Butty´an L., Hubaux, J.P. // Journal of Computer Security – 2004. – 12(3-4) – p. 551-587.
74. Bella G. What is Correctness of Security Protocols? / Bella G. // Journal of universal computer science. - 2008 – vol. 14(12) – p. 2083-2107.
75. Armando A. SAT-based Model-Checking for Security Protocols Analysis / Armando A., Compagna L. // International Journal of Information Security. - 2007 – vol. 6(1) – p. 3-32.

76. Armando A. LTL Model Checking for Security Protocols / Armando, A., Carbone, R., Compagna, L. // AI Communications. - Springer, Heidelberg. – 2007. - p.112-133.
77. Armando A. Model-Checking for Security Protocols Analysis / Armando, A., Compagna, L. // International Journal of Information Security. - 2008 – vol. 7(1) – p. 3-32.
78. Neuman, B.C. An authentication service for computer networks, from IEEE communications magazine / Neuman, B.C., Ts'o, T.: Kerberos // Practical Cryptography for Data Internetworks. - Los Alamitos : IEEE Press,. - 1996.

References for preparation for seminar №1 and №2

79. Pereverzeva Inna Formal Development of Resilient Distributed Systems / Inna Pereverzeva // PhD diss., Turku Centre for Computer Science, Abo Akademi University, Faculty of Science and Engineering, Joukahaisenkatu, Turku, Finland. – 2015.
80. Back R. J. R. From Action Systems to Modular Systems / R. J. R. Back, K. Sere. - 1996 – 17(1) – p. 26-39.
81. Baier C. Principles of Model Checking / C. Baier, J.-P. Katoen // MIT press. - 2008.
82. Bernardeschi C. Analysis of wireless sensor network protocols in dynamic scenarios / C. Bernardeschi, P. Masci, H. Pfeifer // Lecture Notes in Computer Science.- Springer. - 2009. – P. 105-119.
83. Bacherini S. A story about formal methods adoption by a railway signaling manufacturer / S. Bacherini, A. Fantechi, M.Tempestini, and N.Zingoni // Lecture Notes in Computer Science. - Springer. - 2006. – P. 179-189.
84. Barker K. Resilience-based network component importance measures / K. Barker, J. E.Ramirez-Marquez, C. M. Rocco Sanseverino // Reliability Engineering & System Safety. – 2013 – vol. 117 – p. 89-97.
85. Back R. J. R. Stepwise Refinement of Action Systems. Structured Programming / R. J. R. Back, K. Sere // Lecture Notes in Computer Science. – 1991 – 12(1) – p. 17-30.

86. Aziz B. From goal-oriented requirements to event-b specifications / B. Aziz, A. Arenas, J. Bicarregui, Ch. Ponsard, Ph. Massonet // NFM. - 2009. - P. 96-105.
87. Bacherini S. A story about formal methods adoption by a railway signaling manufacturer / S. Bacherini, A. Fantechi, M.Tempestini, N.Zingoni // Lecture Notes in Computer Science. - Springer. - 2006. - P. 179-189.
88. Ball E. Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction / E. Ball, M. Butler // Lecture Notes in Computer Science. - Springer. - 2009. - P. 104-129.
89. Back R. J. R. Decentralization of Process Nets with Centralized Control / R. Kurki-Suonio, R. J. R. Back // Proceedings of the second annual ACM symposium on Principles of distributed computing. - 1983. – P.131-142.
90. Banks Jerry. Principles of simulation. / John Wiley & Sons // Handbook of Simulation. - 2007. – P. 3-30.

CONTENT

Appendix A. Curriculum

DESCRIPTION OF THE MODULE

TITLE OF THE MODULE	Code
Formal and Intellectual Methods for System Security and Resilience	

Teacher(s)	Department
Coordinating: Oksana Pomorova **Others: Sergii Lysenko, Dmytro Medzatyi**	System Programming

Study cycle	Level of the module	Type of the module
Doctoral	A	Full-time tuition

Form of delivery	Duration	Langage(s)
Full-time tuition	One semester	English

Prerequisites	
Prerequisites: Formal Methods; Foundation of Modeling; Computer Networks; Artificial Intelligence; Computer Systems and System Analysis	**Co-requisites (if necessary):**

Credits of the module	Total student workload	Contact hours	Individual work hours
4	108	36	72

Aim of the module (course unit): competences foreseeen by the study programme

The aim of module is to create a knowledge base for formal methods for System Security and Resilience and to provide a prerequisites for practical use of B-method for specifying and designing computer systems and software with formal notation. The study also expands the current research on artificial intelligence in cyber defense. .		
Learning outcomes of module (course unit)	**Teaching/learning methods**	**Assessment methods**
At the end of course, the successful student will be able to: 1. apply Formal Analysis and Design for Security Engineering to industry-related case studies in order to demonstrate the feasibility and effectiveness of the approach in building secure computer systems and software in a provable way	Interactive lectures, Learning in laboratories, Just-in-Time Teaching	Module Evaluation Questionnaire
2. model and analyze the security properties in architecture designs; model security functional and non-functional properties; to use the automated analysis of non-functional properties by formal methods; use a combination of semi-formal UML and formal methods in order to achieve the modelling efficiency provided by UML and the rigorous analysis provided by formal methods; use the model checking and theorem prover as the tools in the analysis of non-functional properties; use of different notations, tailored notations for modelling and analyzing a comprehensive collection of security properties	Interactive lectures, Learning in laboratories, Just-in-Time Teaching	Module Evaluation Questionnaire

in software architectures		
3. use model checkers and theorem provers for verifying that a formal specification satisfies a security property of interest; automatically generate test cases that check source code annotations; automatically construct efficient provably correct code from specifications	Interactive lectures, Learning in laboratories, Just-in-Time Teaching	Module Evaluation Questionnaire
4. use the BAN logic and the authentication of logic in order to verify in the correctness of a protocol; use the process algebra CSP for describing and reasoning about the behaviour of concurrent systems and for reasoning about the high-level interactions and events that may occur during a run of a protocol; take into account the security properties and build methods for assessing the security of a system; use formal methods for the detection of weaknesses and possible attacks; use and apply tools that automatically translate abstract descriptions of security protocols into process-algebraic descriptions that can be analyzed with model checkers	Interactive lectures, Learning in laboratories, Just-in-Time Teaching	Module Evaluation Questionnaire
5. architect of an intelligent system for information security management; build the adaptive and capable systems for discovering and building new knowledge for the information security domain	Interactive lectures, Learning in laboratories, Just-in-Time Teaching	Module Evaluation Questionnaire
6. use the techniques based on Artificial Intelligence for information security	Interactive lectures, Learning in laboratories,	Module Evaluation Questionnaire

management and cyber defense	Just-in-Time Teaching	
7. use and apply the quantitative safety assessment into resilient system development in event-B; apply b-method for merging logical (qualitative) reasoning about resilience of system behaviour; involve the B Method and Event-B in the development of resilient systems	Interactive lectures, Learning in laboratories, Just-in-Time Teaching	Module Evaluation Questionnaire

Themes	Contact work hours							Time and tasks for individual work	
	Lectures	Consultations	Seminars	Practiacl work	Laboratory work	Placements	**Total contact work**	**Individual work**	**Tasks**
1. Formal Analysis and Design for Security Engineering 1.1. Introduction to formal methods. 1.2. Formal Analysis and Design for Security Engineering 1.3. Formal methods for Architecting Secure Software Systems	6				4		**10**	21	
2. Formal Methods for the Analysis of Security Protocols 2.1. Formal Methods for Assuring Security of Computer Networks 2.2. Formal Methods for the Analysis of Security Protocols. Soundness of Formal Encryption 2.3. Formal Methods for the Analysis of Security Protocols. Process Algebras for Studying Security 2.4. A Process Algebra for Reasoning about Quantum Security	8				4		**12**	28	
3. Formal and Intellectual Methods for System Security and Resilience 3.1. Intellectual methods	6		4				**10**	23	

for security 3.2. Methods and Techniques for Formal Development and Quantitative Assessment. Resilient systems 3.3. Formal Development and Quantitative Assessment of Resilient Distributed Systems									
Iš viso	**20**		**4**		**8**		**32**	**72**	

Assessment strategy	**Weight in %**	**Deadlines**	**Assessment criteria**
Lecture activity, including fulfilling special self-tasks	10	7,14	85% – 100% Outstanding work, showing a full grasp of all the questions answered. 70% – 84% Perfect or near perfect answers to a high proportion of the questions answered. There should be a thorough understanding and appreciation of the material. 60% – 69% A very good knowledge of much of the important material, possibly excellent in places, but with a limited account of some significant topics. 50% – 59% There should be a good grasp of several important topics, but with only a limited understanding or ability in places. There may be significant omissions. 45% – 49% Students will show some relevant knowledge of some of the issues involved, but with a good grasp of only a

			minority of the material. Some topics may be answered well, but others will be either omitted or incorrect. 40% – 44% There should be some work of some merit. There may be a few topics answered partly or there may be scattered or perfunctory knowledge across a larger range. 20% – 39% There should be substantial deficiencies, or no answers, across large parts of the topics set, but with a little relevant and correct material in places. 0% – 19% Very little or nothing that is correct and relevant.
Learning in laboratories	30	7,14	85% – 100% An outstanding piece of work, superbly organised and presented, excellent achievement of the objectives, evidence of original thought. 70% – 84% Students will show a thorough understanding and appreciation of the material, producing work without significant error or omission. Objectives achieved well. Excellent organisation and presentation. 60% – 69% Students will show a clear understanding of the issues involved and the work should be well written and well organised. Good work towards the objectives. The exercise should show

			evidence that the student has thought about the topic and has not simply reproduced standard solutions or arguments. 50% – 59% The work should show evidence that the student has a reasonable understanding of the basic material. There may be some signs of weakness, but overall the grasp of the topic should be sound. The presentation and organisation should be reasonably clear, and the objectives should at least be partially achieved. 45% – 49% Students will show some appreciation of the issues involved. The exercise will indicate a basic understanding of the topic, but will not have gone beyond this, and there may well be signs of confusion about more complex material. There should be fair work towards the laboratory work objectives. 40% – 44% There should be some work towards the laboratory work objectives, but significant issues are likely to be neglected, and there will be little or no appreciation of the complexity of the problem. 20% – 39% The work may contain some correct and relevant material, but most issues are neglected or are covered incorrectly. There should be some signs of appreciation of the laboratory

			work requirements. 0% – 19% Very little or nothing that is correct and relevant and no real appreciation of the laboratory work requirements.
Module Evaluation Quest	60	8,16	The score corresponds to the percentage of correct answers to the test questions

Author	Year of issue	Title	No of periodical or volume	Place of printing. Printing house or intrenet link
Compulsory literature				
Dr. Hubert Garavel	2013	Formal Methods for Safe and Secure Computers Systems		Federal Office for Information Security
P. Popov, O. Netkachov, K. Salako	2014	Model-based evaluation of the resilience of critical infrastructures under cyber attacks	№1. – p. 231-243	International Conference on Critical Information Infrastructures Security
Vain J.	2007	Formal Techniques for Networked and Distributed Systems	№4574. – p. 364-373.	FORTE 2007: 27th IFIP WG 6.1 International Conference, Tallinn, Estonia, June 27-29, 2007, Proceedings, Springer Science & Business Media
J. Vain, E. Halling, G. Kanter, A. Anier, D. Pal	2016	Model-Based Testing of Real-Time Distributed Systems	№2. – p. 272-286.	International Baltic Conference on Databases and Information

				Systems
T. Tagarev, H. Bucur-Marcu, P. Flur	2009	Defence Management: An Introduction	212 p.	Geneva : DCAF, Geneva Centre,
S. Russo, G. Carrozza, R. Pietrantuono	2014	Defect analysis in mission-critical software systems: a detailed investigation	№1699. – p. 22-49.	Software: Evolution and Process
S. Russo, D. Cotroneo, R. Pietrantuono	2015	RELAI testing: a technique to assess and improve software reliability	№42. – p. 452-475	IEEE Transaction on Software Enginering.
V. Kharchenko, A. Tarasyuk, A. Gorbenko	2015	Principles of Formal Methods Integration for Development Fault-Tolerant Systems: Event-B and FME (C)	№3. – p. 423-429.	Journal of Computing
Kharchenko, B. Volochiy, O. Mulyak, L. Ozirkovskyi	2016	Automation of Quantitative Requirements Determination to Software Reliability of Safety Critical NPP I&C systems	№6. – p. 337-346	Second International Symposium on Stochastic Models in Reliability
O. Tarasyiuk, A. Gorbenko	2009	Formal method for the development of the critical software		Kharkiv: Nationa Aerospace univ «Kharkiv Aviation Institure»
Romanovsky A.	2012	Deployment of Formal Methods in Industry: the Legacy of the FP7 ICT DEPLOY Integrated Project",		Newcastle University, Computing Science Newcastle upon Tyne
Laprie, J-C.	2008	From dependability to		38th Annual IEEE/IFIP

		resilience		International Conference on Dependable Systems and networks
Reder, L., Day, J., Ingham, M., Murray, R., and Williams, B.	2012	Engineering Resilient Space Systems Introduction to Short Course		
Enn Tyugu	2011	Artificial Intelligence in Cyber Defense		3rd International Conference on Cyber Conflict (Tallinn, Estonia)
Lirong Dai and Kendra Cooper	2007	A Survey of Modelling and Analysis Approaches for Architecting Secure Software Systems		International Journal of Network Security
Constance L. Heitmeyer, Myla M. Archer, Elizabeth I. Leonard and John D. McLean	2008	Applying Formal Methods to a Certifiably Secure Software System		SOFTWARE ENGINEERING
Pedro Miguel dos Santos Alves Madeira Adão	2006	Formal Methods for the Analysis of Security Protocols		PhD diss., INSTITUTO SUPERIOR TÉCNICO
Bruno Blanchet	2011	Using Horn Clauses for Analyzing Security Protocols		Formal Models and Techniques for Analyzing Security Protocols
Oksana Pomorova, Oleg Savenko, Sergii Lysenko, Andrii Kryshchuk	2013	Multi-Agent Based Approach for Botnet Detection in a Corporate Area Network Using Fuzzy Logic		Computer Networks Communications in Computer and Information Science

Lysenko S., Savenko O., A. Kryshchuk, Y. Klyots	2013	Botnet detection technique for corporate area network		Proceedings of the 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems, Berlin, DE, IEEE
Inna Pereverzeva	2015	Formal Development of Resilient Distributed Systems		PhD diss., Turku Centre for Computer Science, Abo Akademi University, Faculty of Science and Engineering, Joukahaisenkatu, Turku, Finland
Inna Pereverzeva, Elena Troubitsyna, Linas Laibinis, Mats Brorsson, Luis Miguel Pinho	2012	Formal Goal-Oriented Development of Resilient MAS in Event-B,		Proceedings of 17th International Conference on Reliable Software Technologies (Ada-Europe), Springer-Verlag Berlin Heidelberg, 2012.
Almeida J.B., Frade M.J., Pinto, J.S., Melo de Sousa S.	2011	Rigorous Software Development. An Introduction to Program Verification		Springer
Véronique Cortier, Steve Kremer	2011	Formal Models and Techniques for Analyzing Security Protocols	Vol.5	IOS Press
Riham Hassan Abdel-Moneim Mansour	2009	Formal Analysis and Design for Engineering Security (FADES)		Blacksburg, Virginia

Additional literature				
Bruce Christianson, Bruno Crispo, James A.Malcolm, Michael Roe	2006	Security Protocols 14th International Workshop Cambridge, UK, March 27-29		Springer Berlin Heidelberg New York ISBN-13 978-3-642-04903-3
J.-C. Laprie	2005	Resilience for the Scalability of Dependability,		Fourth IEEE International Symposium on Network Computing and Applications. IEEE,
Tarasyuk, Anton and Troubitsyna, Elena and Laibinis, Linas, In:,	2012	Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B.		Lecture Notes in Computer Science Springer

Syllabus

MODULE1 Formal Analysis and Design for Security Engineering

1 TOPIC 1. Introduction to FormalMethods

1.1 What are Formal Methods?

1.2 The Nature of Formal Methods

1.3 Benefits in the use of Formal Methods

2 TOPIC 2. Formal Analysis and Design for Security Engineering

2.1 Knowledge Acquisition for Automated Specifications – Goal-Oriented Requirements of the Security Engineering

2.2 Goal-Oriented Requirements of the Security Engineering

2.4 The B Method

2.3 Formal Analysis and Design for Security Engineering

3.3 FADES Tool Support

3 TOPIC 2. Formal methods for Architecting Secure Software Systems

1.1. Systematical security engineering into software applications

1.2 Semi-formal Security Modelling and Analysis Approaches

1.3 MAC-UML Framework. SecureUML

1.4 Separating Modelling of Application and Security Concerns

1.5 Formal Security Modelling and Analysis Approaches

1.6 Integrated Semi-formal and formal Modelling and Analysis Approaches

LAB1 Formal Analysis and Design for Security Engineering. Demonstration with Case Studies. The Spy Network Case Study

4.1.1 Case Study Preliminary Problem Statement

4.1.2 Elaborating Security Requirements with KAOS

4.1.2.1 Integrity Goals

4.1.2.2 Confidentiality Goals

4.1.2.3 Authentication Goals

4.1.2.4 Availability Goals

4.1.2.5 Access Control Goals

4.1.3 Analysis and Resolution of Obstacles and Conflicts for Security Goals

4.1.3.1 Generating Obstacle to the Goal RevelationIntegrity
4.1.3.2 Resolving Obstacles to the Goal RevelationIntegrity
4.1.4 Transforming the Spy Network Security Goal Graph to B
4.1.5 Derivation of Design and Implementation
4.1.6 Acceptance Testing
4.1.7 Security Specifications Changes

LAB 2. Formal Methods for a Certifiable Secure Software System
2.1. Code verification process
2.2 Formal foundations for a certifiably secure software system
2.3 Applying formal Techniques to Other Security Properties

MODULE 2 . Formal Methods for the Analysis of Security Protocols.

4 TOPIC 1. Formal Methods for Assuring Security of Computer Networks
1.1 Formal Methods and Security of Computer Networks
1.2 Needham–Schroeder protocol
1.3 Tools for formal methods
1.4 Model–based software development
1.5 Principals of security.
1.6 Key security properties
1.7 Assessing security protocols
1.8. Needham–Schroeder public–key protocol.BAN logic

5 TOPIC 2. Formal Methods for the Analysis of Security Protocols. Soundness of Formal Encryption
4.1 The Abadi-Rogaway Logics of Formal Encryption. Process Algebras for Security. Quantum Security
4.2 The Abadi-Rogaway Soundness Theorem
4.3 Soundness in the Presence of Key-Cycles
4.4 Partial Leakage of Information
4.5 Information-Theoretic Interpretations: Soundness and Completeness for One-Time Pad
4.6 General Treatment for Symmetric Encryption

6 TOPIC 3. Formal Methods for the Analysis of Security Protocols. Process Algebras for Studying Security
3.1 Low-Level Target Model
3.2 A Distributed Calculus with Principals and Authentication

3.2.1 Syntax and Informal Semantics
3.2.2 Operational Semantics
3.2.3 An Abstract Machine for Local Reductions
3.3 High-Level Equivalences and Safety
3.4 Applications
3.4.1 Anonymous Forwarders
3.4.2 Electronic Payment Protocol
3.4.3 Initialisation
3.5 A Concrete Implementation
3.5.1 Implementation of Machines

7 TOPIC 3. A Process Algebra for Reasoning about Quantum Security

4.1 Process Algebra
4.1.1 Quantum polynomial machines
4.1.2 Process algebra
4.1.3 Semantics
4.1.4 Observations and observational equivalence
4.2 Emulation and Composition Theorem
4.3 Quantum Zero-Knowledge Proofs

LAB 4. Using Horn Clauses for Analyzing Security Protocols

LAB 5 Validating Security Protocols under the General Attacker

MODULE 3 Formal and Intellectual Methods for System Security and Resilience

8 TOPIC 1. Intellectual methods for security

1.1 Artificial Intelligence Techniques Applied to Intrusion Detection.
1.2 Multi-agent based approach of botnet detection in computer systems
1.3 Technique for bots detection which use polymorphic code

9 TOPIC 2. Methods and Techniques for Formal Development and Quantitative Assessment. Resilient systems

2.1 Resilience and Dependability: Basic Definitions. Goal-Based Development
2.2 Development Methodologies
2.3 Event-B Method
2.4 Quantitative Assessment

2.5 PRISM model checker

2.6 Discrete-event simulation

10 TOPIC 3. Formal Development and Quantitative Assessment of Resilient Distributed Systems

3.1 Resilience-Explicit Development Based on Functional Decomposition

3.2 Modelling Component Interactions of the Resilient System with Multi-Agent Framework

3.3 Goal-Oriented Modelling of Resilient Systems

3.4 Pattern-Based Formal Development of Resilient MAS

3.5 Formal Goal-Oriented Reasoning About Resilient Reconfigurable MAS

3.6 Modelling and Assessment of Resilient Architectures

Seminar №1. Formal Goal-Oriented Development of Resilient MAS in Event-B

Seminar№2. Formal Modelling of Resilient Data Storage in Cloud.

Oksana Pomorova,
Sergii Lysenko,
Dmytro Medzatyi

Formal and Intellectual Methods for System Security and Resilience

Practicum

Editor Kharchenko V.

Зв. план, 2017
Підписаний до друку 20.03.2017 Формат 60x84 1/16. Папір офс. №2. Офс. друк.
Умов. друк. арк. 21,89. Уч.-вид. л. 22,31. Наклад 100 прим.
Замовлення 2/2. Ціна вільна

Національний аерокосмічний університет ім. М. Є. Жуковського "Харківський авіаційний інститут"
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu