School of Computing Science,
University of Newcastle upon Tyne

Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems (REFT 2005)

Michael Butler, Cliff Jones, Alexander Romanovsky, and
Elena Troubitsyna

Technical Report Series

CS-TR-915

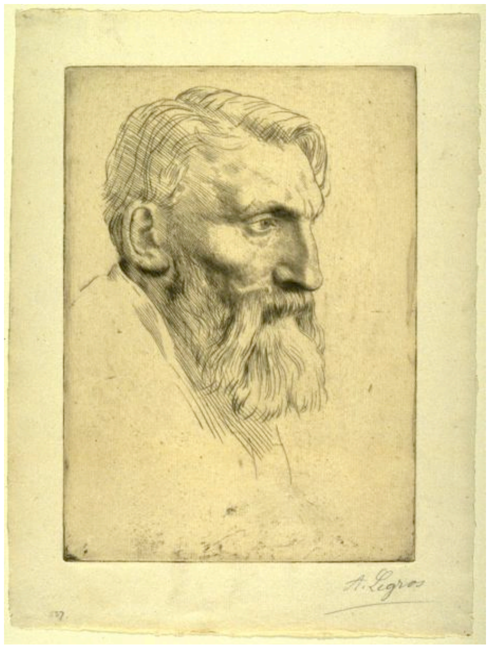
June 2005

Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems (REFT 2005)

at the 13th International Symposium of Formal Methods 2005

Newcastle upon Tyne, UK
July 19, 2005

This workshop is organised by the partners of FP6 IST RODIN
Rigorous Open Development Environment for Complex Systems



WORKSHOP ORGANISERS
Michael Butler (University of Southampton)
Cliff Jones (University of Newcastle upon Tyne)
Alexander Romanovsky (University of Newcastle upon Tyne)
Elena Troubitsyna (Aabo Akademi)

http://rodin.cs.ncl.ac.uk/

Preface

This report contains the proceedings of the 2005 workshop on Rigorous Engineering of Fault Tolerant Systems (REFT 2005) held in conjunction with the Formal Methods 2005 conference in Newcastle upon Tyne, UK. The aim of this one day workshop is to bring together researchers who are interested in the application of rigorous design techniques to the development of fault tolerant software based systems. Fault tolerance design techniques are essential for increasing the dependability of complex systems. It is our belief that such techniques need to be designed into systems in a rigorous and principled way. It is also our belief that the use of formal methods is essential for rigorous engineering of any complex system. Through abstraction, refinement and proof, formal methods provide design techniques that support clear thinking as well rigorous validation and verification. Good tool support is also required to support the industrial application of these design techniques.

The nature of scientific research is such that people tend to belong to communities with common interests and usually there is insufficient dialogue between communities who may have much to offer each other. In organising this workshop we sought contributions from the fault tolerance community and the formal methods community. Our hope is that the formal methods people can learn more about, and perhaps be fired up by, challenging issues in fault tolerant design. Likewise, we hope that researchers on fault tolerance can understand better how formal methods could improve the way in which their techniques are developed and applied.

The REFT 2005 workshop was organised by the partners of FP6 IST RODIN (Rigorous Open Development Environment for Complex Systems). Rigorous design of fault tolerant systems is a major theme of the RODIN project. In organising this workshop we are aiming to build a network of researchers from the wider community to promote integration of the dependability and formal methods research.

We were delighted with the quality and relevance of the paper submissions that we received for the workshop. Approximately half the papers are from members of the RODIN project while the other half are from the wider community.

We have several papers from fault tolerance researchers, several from formal methods researchers and several that involve researchers in both communities. It is encouraging to see that many of the papers are addressing software based systems that impact peoples' everyday lives such as communications systems, mobile services, control systems, medical devices and business transactions. We hope that you enjoy reading these proceedings and encourage you to contribute to our aim of closer collaboration between dependability and formal methods research.

Michael Butler (University of Southampton)
Cliff Jones (University of Newcastle upon Tyne)
Alexander Romanovsky (University of Newcastle upon Tyne)
Elena Troubitsyna (Aabo Akademi)

Workshop papers

Ian Hayes. Using domain models to specify systems (invited talk)	1
Linas Laibinis, Elena Troubitsyna, Sari Leppanen, Johan Lilius, Qaiser Malik. Formal Service-Oriented Development of Fault Tolerant Communicating Systems	2
Colin Snook, Michael Poppleton, Ian Johnson. Towards a methodology for rigorous development of generic requirements patterns	12
C. Ponsard, P. Massonet, J.F. Molderez Analyzing Fault-Tolerant Systems with FAUST (demonstration)	22
Shmuel Katz. Rigorous Fault Tolerance Using Aspects and Formal Methods (invited presentation)	32
Alfredo Capozucca, Nicolas Guelfi, Patrizio Pelliccione. The Fault-Tolerant Insulin Pump Therapy	33
Thomas Wilson, Savi Maharaj, Robert Clark. Omnibus: A clean language and supporting tool for integrating different assertion-based verification techniques (short presentation)	43
Alexei Iliasov, Linas Laibinis, Alexander Romanovsky, Elena Troubitsyna. Towards Formal Development of Mobile Location-based Systems	53
Joey Coleman, Cliff Jones. Examples of how to Determine the Specifications of Control Systems	65
Anatoliy Gorbenko, Vyacheslav Kharchenko, Olga Tarasyuk. FMEA- technique of Web Services Analysis and Dependability Ensuring	74
Dubravka Ilic, Elena Troubitsyna. Modelling Fault Tolerance of Transient Faults	84
Divakar Yadav, Michael Butler. Application of Event B to Global Causal Ordering for Fault Tolerant Transactions	93
Patrice Chalin. Are Practitioners Writing Contracts?	103
Fernando Castor Filho, Patrick Henrique da S.Brito, Cecilia Mary F.Rubria. Modeling and Analysis of Architectural Exceptions	112
Joey Coleman. Examining BPEL's Compensation Construct (short presentation)	122
Alexei Iliasov, Victor Khomenko, Maciej Koutny, Alexander Romanovsky. On Specification and Verification of Location-based Fault Tolerant Mobile Systems	129
Victor Khomenko. Shortest Violation Traces in Model Checking Based on Petri Net Unfoldings and SAT (short presentation)	141

Using domain models to specify systems

Invited Talk

Ian Hayes

School of Information Technology and Electrical Engineering
The University of Queensland
Australia

Abstract: In order to specify a control system one needs a model of the **domain** being controlled including its **interface** to the controlling **machine**. It should be adequate to formally specify:

- the overall system's **required behaviour** (1),
- the **assumptions** the machine can rely on about the domain's (normal) behaviour (2), and
- the **constraints** on the way the domain may be controlled via its interface.

To accommodate fault-tolerance one also needs to be able to formally specify:

- **hazardous** behaviour of the system (to be avoided),
- possible **misbehaviour** of the domain -- faults or failure modes -- this weakens the assumptions (2),
- allowable responses to faults -- this weakens (1), and
- **healthy** behaviour of the domain to allow checks to be made on the domain's behaviour -- this should imply the assumptions (2).

Choice of an adequate level of abstraction for the domain model is essential (and difficult). It should allow the specification of the above characteristics without including extraneous characteristics. For this an engineer with domain experience is typically required.

This work is conducted in cooperation with Michael Jackson and Cliff Jones.

Formal Service-Oriented Development of Fault Tolerant Communicating Systems

Linas Laibinis*, Elena Troubitsyna*, Sari Leppänen**, Johan Lilius*, Qaisar Malik*

*Åbo Akademi University, Department of Computer Science, Lemminkaisenkatu 14 A, 20520, Turku, Finland
** Nokia Research Center, Mobile Networks Laboratory, P.O. Box 407, 00045, Helsinki, Finland

1. Introduction

Majority of engineering methods for building complex systems is based on system decomposition. In the software engineering the decomposition-based development methods are often referred to as the service-oriented methods. The notion of a *service* provides a convenient mechanism for modelling and reasoning about system interactions and functionality.

In the telecommunicating systems, a service is usually understood as a coherent piece of functionality that the system delivers to its users. Since telecommunicating systems are distributed by their nature, a service is usually provided by several collaborating service components. Often communication between service components relies on an unreliable media, such as, e.g., radio-based mobile network. Hence communication failures are an intrinsic part of system behaviour. Therefore, the correct service provision is unfeasible without integrating the fault tolerance mechanisms in the system design.

In this paper we propose a formal approach to service-oriented development of fault tolerant communicating distributed systems. Our approach is based on formalization of the service-oriented methodology Lyra [4] developed in the Nokia Research Center. The design flow of Lyra is based on concepts of decomposition and preservation of the externally observable behaviour. The system behaviour is modularised and organized into hierarchical layers according to the external communication and related interfaces so that the distributed network architecture can be derived from the functional system requirements via a number of model transformations. This approach coincides with the stepwise refinement paradigm adopted in the B Method [1].

In this paper we describe our work on the formalizing Lyra in B. We propose general specification and development patterns according to which the services can be specified and decomposed into communicating service components. The patterns generalise the existing practice of the communicating system engineering. Hence our approach provides the basis for automating the process of development of communicating systems correct by construction. It is illustrated by a case study – development of a Third Generation Partnership Project (3GPP) positioning system.

2. Overview of Lyra

The Lyra design method consists of four phases corresponding to the classical design phases: Service Specification, Service Decomposition, Service Distribution and Service Implementation. These phases correspond also to the conventional phases of standardization [2]. In the Service Specification phase we define the services provided by the system (standardization Phase 1). In the Service Decomposition phase we specify the functional architecture of each the system level service (standardization Phase 2). In the Service Distribution phase logical entities of the functional architecture, i.e. service components, are distributed over a given network architecture and signalling protocols are defined for communication between the network elements (standardization Phase 3). In the Service Implementation phase we adjust the functionality to the target environment. The program code for a specific platform is generated automatically from the resulting implementation.

Next we describe the general idea of the methodology with a running example. We model part of a Third Generation Partnership Project (3GPP) positioning system [7,8]. The positioning system provides positioning services to calculate the physical location of a given user equipment (UE) in a Universal Mobile Telecommunication System (UMTS) network. We focus on Position Calculation Application Part (PCAP) – a part of the positioning system allowing communication in the Radio Access Network (RAN). PCAP manages the communication between the Radio Network Controller (RNC) and the Stand-alone Assisted Global Positioning System Serving Mobile Location Centre (SAS) network elements. The functional requirements for the RNC-SAS communication have been specified in [7,8].

The Service Specification phase defines the service in terms of its communication with the service consumer as shown in Fig. 1. The service consumer requests the positioning calculation and receives the result of the service execution via the provided service access point (the upward interface). At this development stage we abstract away from the details of the position computation and merely observe that the service execution can result either in the position calculated with the requested accuracy or in a failure.

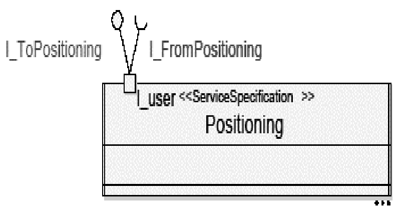


Fig 1. Service specification

At the next stage of the development process – Service Decomposition we take into account that a service is provided in co-operation with service components. The initial model presented in Fig.1 is augmented with the used service access points (the downward interfaces) via which the communication with service components is conducted. The model obtained as a result of the Service Decomposition phase is presented in Fig.2.

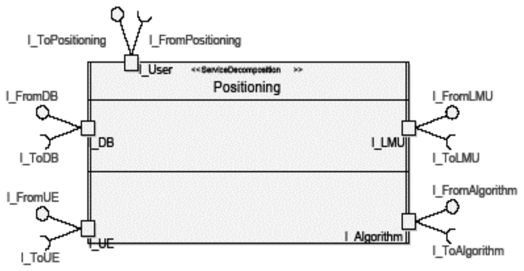


Fig 2. Service decomposition

At the Service Decomposition phase we also design the functional architecture of the service, as shown in Fig 3. Usually the functional architecture is constructed according to the following pattern: a service director orchestrates the service execution by requesting certain services from service components. For instance, to execute the position calculation service, the service director first requests an approximate location of the UE from the network database, then it requests the UE to send additional radio measurements, then it requests several local measurement units (LMU) to provide some local measurements, and finally the data collected during all these stages are sent to a location algorithm server which invokes a certain algorithm for position calculation. After executing the algorithm, the calculated position is returned to the service director. Let us observe that any of the requested components can fail (either because of communication or some other failure). When a request to a service component fails, the service director diagnoses the failure and decides on the fault tolerance measures to be undertaken.

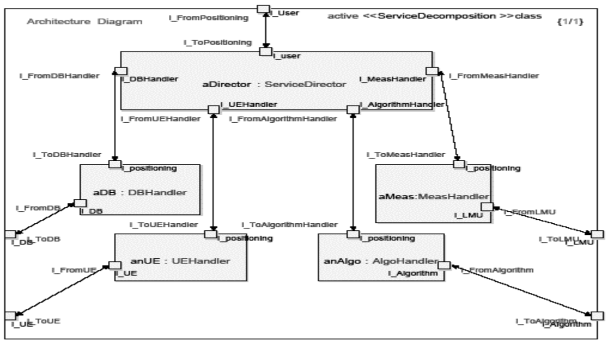


Fig 3. Functional architecture

To manage complexity of communicating systems, at the Service Decomposition phase the communication between components still remains on a virtual level – the realistic communication protocols are introduced upon completing the next (Service Distribution) stage. At the Service Distribution phase we map the functional system architecture to the platform architecture. For instance, in our example we decompose the system in such a way that communication with the network database and UE is performed by the service director allocated on RNC, while communication with LMU devices and the algorithm server is performed by the service director allocated on SAS. The service directors communicate via a certain (predefined by PCAP) protocol. The result of service distribution is shown in Fig. 4.

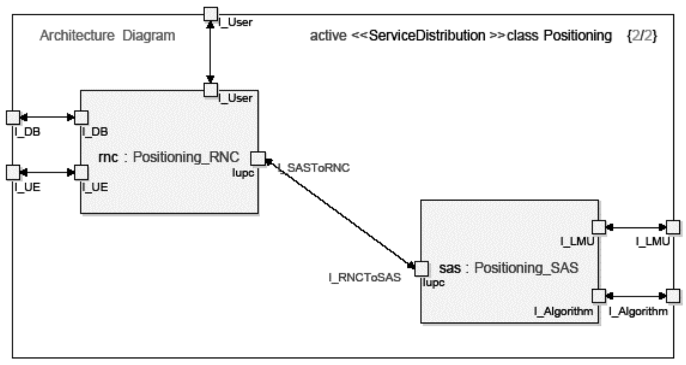


Fig 4. Platform architecture

In the final (Service Implementation) phase we adjust the model to fit a specific platform. We omit the detailed discussion of this stage. In the next section we give a brief introduction into our formal framework – the B Method, which we will use to formalize the development flow described above.

3. The B Method

The B Method [1] (further referred to as B) is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [5]. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [6], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

The development methodology adopted by B is based on stepwise refinement [1]. While developing a system by refinement, we start from an abstract formal specification and transform it into an implementable program by a number of correctness preserving steps, called *refinements*. A formal specification is a mathematical model of the required behaviour of a (part of) system. In B a specification is represented by a set of modules, called Abstract Machines. An abstract machine encapsulates state and operations of the specification and as a concept is similar to a module or a package.

Each machine is uniquely identified by its name. The state variables of the machine are declared in the VARIABLES clause and initialised in the INITIALISATION clause. The variables in B are strongly typed by constraining predicates of the INVARIANT clause. All types in B are represented by non-empty sets.

The operations of the machine are defined in the OPERATIONS clause. In this paper we use Event B extension of the B Method. The operations in Event B are described as guarded statements of the form SELECT cond THEN body END. Here cond is a state predicate, and body is a B statement. If cond is satisfied, the behaviour of the guarded operations corresponds to the execution of their bodies. However, if cond is false, then the execution of the corresponding operation is suspended, i.e., the operation is in waiting mode until cond becomes true.

B statements that we are using to describe a state change in operations have the following syntax:

```
S  ==  x := e  |  IF cond THEN S1 ELSE S2 END | S1 ; S2  |  x :: T  |
       S1 || S2 | ANY z WHERE cond THEN S END  |  ...
```

The first three constructs – assignment, conditional statement and sequential composition (used only in refinements) have the standard meaning. The remaining constructs allow us to model nondeterministic or parallel behaviour in a specification. Usually they are not implementable so they have to be refined (replaced) with executable constructs at some point of program development. The detailed description of the B statements can be found elsewhere [1].

The B method provides us with mechanisms for structuring the system architecture by modularisation. A module is described as a machine. The modules can be composed by means of several mechanisms providing different forms of encapsulation. For instance, if the machine C INCLUDES the machine D then all variables and operations of D are visible in C. However, to guarantee internal consistency (and hence independent verification and reuse) of D, the machine C can change the variables of D only via the operations of D. In addition, the invariant properties of D are included into the invariant of C.

To illustrate basic principles of specifying and refining in B, next we present our approach to formal service-oriented development.

4. Formal Service-Oriented Development

We start to formalize the Lyra development by creating a specification pattern modelling a communicating component. This pattern is used throughout the entire development process. The pattern is called Abstract Communicating Component (ACC). ACC consists of a "kernel", i.e., the provided functionality, called Abstract

Calculating Machine (ACAM), and a "communication wrapper", i.e., the communication channels via which data are supplied to and consumed from the component, called Abstract Communicating Machine (ACM).

The specification of an abstract communicating component (ACC) consists of operations specifying ACM and ACAM. The variables inp_chan and out_chan model the input and output channels. The environment places requests for the service by assigning to inp_chan and receives the results of the service via out_chan. Data transferred to and from ACC are modelled abstractly. We reserve the abstract constant NIL to model the absence of data, i.e., the empty channel. The variables input and output are one-place data buffers internal to the ACC. The variable input stores the data read from inp_chan. It is used as a temporal data storage in calculating the required service. The variable output stores the final result of calculations which is consequently put into the output channel out_chan for the server consumer.

The operations env_write and env_read model the behaviour of the service consumer: placing the request to execute a service and reading the results of its execution. The operation read models reading the service request by ACC from the input channel. The symmetric operation write writes the results of service provision into the output channel. These operations specify the "communication wrapper" (i.e., ACM) part of ACC.

In the initial specification, ACAM is modelled abstractly by the operation calculate, which non-deterministically assigns the variable output either the result of a successful service provision or a failure. The machine ACC (presented below) specifies the described behaviour instantiated for the position calculation case study.

```
MACHINE  ACC

VARIABLES
  inp_chan, input, out_chan, output

INVARIANT
  inp_chan : INPUT_DATA  &
  input : INPUT_DATA  &
  out_chan : POS_DATA  &
  output : POS_DATA

INITIALISATION
  inp_chan, input := INP_NIL, INP_NIL ||
  out_chan, output := POS_NIL, POS_NIL

OPERATIONS

env_write =
  SELECT inp_chan = INPUT_NIL
  THEN
    inp_chan :: INPUT_DATA - {INPUT_NIL}
  END;

read =
  SELECT not(inp_chan = INP_NIL) &
            (input = INP_NIL)
  THEN
    input,inp_chan := inp_chan,INP_NIL
  END;

calculate =
  SELECT not(input = INP_NIL) &
            (output = POS_NIL)
  THEN
    CHOICE
      output ::
          POS_DATA - {POS_NIL,POS_FAIL}
    OR
      output := POS_FAIL
    END ||
    input := INP_NIL
  END;

write =
  SELECT not(output = POS_NIL) &
(out_chan = POS_NIL)
  THEN
    out_chan,output := output,POS_NIL
  END;

env_read =
  SELECT not(out_chan = POS_NIL)
  THEN
    out_chan := POS_NIL
  END

END
```

The next phases in the Lyra development are the functional and the platform-based decomposition. In our approach they correspond to two consequent refinement steps – first refining the algorithmic part of ACC, i.e. ACAM, and then introducing the peer entities distributed over the given platform. The result of the first refinement is graphically represented in Fig.5.

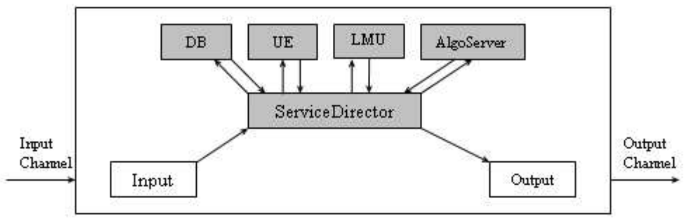


Fig.5. Functional decomposition of ACAM

ACAM is refined by introducing the service director modelled by the operation director which orchestrates the execution of the whole positioning service, and the operations db, ue, lmu, and alg modelling execution of the corresponding service components.

Basically we decompose ACAM to model stages of the positioning service. Additionally, we introduce the variables that model results obtained at these intermediate stages from the corresponding service components. At this refinement step, these variables are non-deterministically updated in the operation director. We model not only successful execution of the intermediate stages by the service components but also possible failures. Moreover, we abstractly model error recovery – upon detecting an error, the service director can retry (up to the predefined number of attempts) to execute a certain stage of the service. However, if error recovery fails, the service director terminates the service execution and returns the error as the final result. The refined specification of the ACC – ACC1 instantiated for the positioning service is presented in Fig.6.

The second refinement that we perform models the mapping of the functional decomposition into the given target platform. We introduce communication with the service components into the specification of the service director. The service components are specified according to the proposed pattern ACC. The service director plays now a role of the service consumer for the service components. Namely, it sends the requests to execute the services required for the corresponding stages and receives the obtained results. Let us observe that at this refinement step we replace non-deterministic updates to the variables storing the results of the intermediate stages by assigning them the values obtained from the communication channels. The graphical representation of this stage for the positioning system is given in Fig.7.

```
REFINEMENT  ACC1

REFINES  ACC

VARIABLES
  curr_service, handling_flag …

INVARIANT
  curr_service : SERVICE  &
  handling_flag : BOOL & …

INITIALISATION
  curr_service, handling_flag := SD,FALSE
  || …

OPERATIONS

env_write =  …

read = …

db =
  SELECT curr_service = DB
  THEN
    handling_flag := TRUE
  END;

ue = …

lmu = …

alg = …

director =
  SELECT handling_flag = TRUE
  THEN
    IF curr_service = SD
    THEN
      curr_service := DB
    ELSIF curr_service = DB
    THEN
     dbdata :: DB_DATA-{DB_NIL};
      IF DB_Eval(dbdata) = OK
      THEN
        curr_service := UE
      ELSIF DB_Eval(dbdata) = RECOV &
              (n_db > 0)
      THEN
        n_db := n_db-1
      ELSE
        posdata,curr_service :=
                    POS_FAIL,CALC
      END
    ELSIF curr_service = UE …
    ELSIF curr_service = LMU …
    ELSIF curr_service = ALG …
    END ||
    handling_flag := FALSE
  END;

calculate =
  SELECT (curr_service=CALC) & …
  THEN
    output,input := posdata,INP_NIL ||
    curr_service := SD
  END;

write = …

env_read =  …

END
```

Fig.6. ACC1 refinement

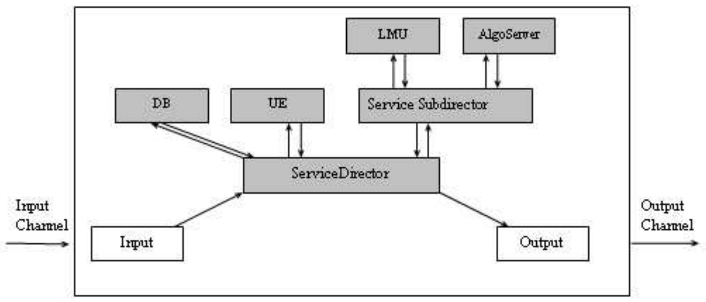


Fig.7. Service distribution

The excerpt from the refinement of ACC1 – the refinement machine ACC2 is given below.

```
director =
  SELECT handling_flag = TRUE &
        (((curr_service = DB) & not(db_out_chan = DB_NIL)) or ...)
  THEN
    IF curr_service = SD
    THEN
      curr_service := DB
    ELSIF curr_service = DB
    THEN
      dbdata <-- db_read_ochan;
      IF DB_Eval(dbdata) = OK
      THEN
        curr_service := UE
      ELSIF DB_Eval(dbdata) = RECOV & (n_db > 0)
      THEN
        n_db := n_db-1
      ELSE
        posdata,curr_service := POS_FAIL,CALC
      END
    ELSIF curr_service = UE ...
    ELSIF curr_service = ALG
    THEN
      posdata <-- sas_read_ochan; ...
    END ||
    handling_flag := FALSE
  END;
```

At the consequent refinement steps we will focus on particular service components and refine them (in the way described above) until the desired level of granularity is obtained. Once all external service components are in place, we can further decompose their specifications by separating their *ACM* and *ACAM* parts. Such decomposition will allow us to concentrate on the communicational parts of the respective components and further refine them by introducing details of required concrete communication protocols.

5. Conclusions

In this paper we proposed an approach to formal modelling of communicating distributed systems. We derived the specification and refinement patterns that can be used to automate the development of such systems. The patterns define the formal semantics of UML diagrams usually used in the development process. Hence UML modelling can be used as a syntactic sugaring of the formal development. Because of the wide acceptance of UML in industry our approach can therefore be easily integrated into existing development practice.

In this paper we demonstrated how to model faulty behaviour and fault tolerance features of communicating systems. Unreliable communication is an intrinsic part of communicating distributed systems. Hence addressing the fault tolerance issues in the development process is an important merit of the proposed approach.

The formalisation of the UML-based development of communicating distributed systems has been undertaken in the Lyra approach. Lyra is based on model checking and enables reasoning about preservation of the externally observable behaviour throughout the development process. However, the model checking techniques are prone to the state explosion problem since telecommunicating systems tend to be large and data intensive. Our approach helps to overcome this limitation.

As a future work, we will continue to develop the proposed approach to address issues of concurrency and verification of the temporal properties of communication protocols between network elements. Moreover, we are planning to develop a tool support to automate the proposed approach.

Acknowledgements
The work reported in this paper is financially supported by IST-511599 EU Project RODIN.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. ITU-T. Rec. I.30 (1993). Method for characterization of telecommunication services supported by an ISDN and network capabilities of an ISDN.
3. L.Laibinis and E.Troubitsyna. *Fault Tolerance in a Layered Architecture: A General Specification Pattern in B.* Proceedings of 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), Beijing, China, September 2004. IEEE Press, pp.346-355.
4. S.Leppänen, M.Turunen, and I.Oliver. *Application Driven Methodology for Development of Communicating Systems*. FDL'04, Forum on Specification and Design Languages. Lille, France, September 2004.
5. MATISSE Handbook for Correct Systems Construction. 2003. http://www.esil.univ-mrs.fr/~spc/matisse/Handbook/
6. Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 2001. Available at http://www.atelierb.societe.com/index uk.html
7. 3GPP. Technical specification 25.305: Stage 2 functional specification of UE positioning in UTRAN. See http://www.3gpp.org/ftp/Specs/html-info/25305.htm
8. 3GPP. Technical specification 25.453: UTRAN Iupc interface positioning calculation application part (pcap) signalling. See http://www.3gpp.org/ftp/Specs/html-info/25453.htm

Towards a methodology for rigorous development of generic requirements patterns[1]

Colin Snook[1], Michael Poppleton[1], and Ian Johnson[2]

[1] School of Electronics and Computer Science,
University of Southampton,
SO17 1BJ, UK,
cfs,mrp@ecs.soton.ac.uk
[2] AT Engine Controls, Portsmouth, UK
ijohnson@atenginecontrols.com

Abstract. We present work in progress on a methodology for the engineering, validation and verification of generic requirements using domain engineering and formal methods. The need to develop a generic requirement set for subsequent system instantiation is complicated by the addition of the high levels of verification demanded by safety-critical domains such as avionics. We consider the failure detection and management function for engine control systems as an application domain where product line engineering is useful. The methodology produces a generic requirement set in our, UML based, formal notation, UML-B. The formal verification both of the generic requirement set, and of a particular application, is achieved via translation to the formal specification language, B, using our U2B and ProB tools.

Introduction

The notion of software *product line* (also known as *system family*) engineering became well established [14], after Parnas' proposal [18] in the 70's of information hiding and modularization as techniques that would support the handling of program families. Product line engineering arises where multiple variants of essentially the same software system are required, to meet a variety of platform, functional, or other requirements. This kind of generic systems engineering is well known in the avionics industry; e.g. [12, 10] describe the reuse of generic sets of requirements in engine control and flight control systems.

Domain analysis and object oriented frameworks are among numerous solutions proposed to product line technology. In Domain-Specific Software Architecture [23] for example, the domain engineering of a set of general, domain-specific requirements for the product line is followed by its successive refinement, in a series of system engineering cycles, into specific product requirements. On the other hand [11] describes the Object-Oriented Framework as a "a reusable, semi-complete application that can be specialized to produce custom applications". Here the domain engineering pro-

[1] This work is part of the EU funded research project: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

duces an object-oriented model that must be instantiated, in some systematic way, for each specific product required. In this work we combine object-oriented and formal techniques and tools in domain and product line engineering.

It is widely recognized that formal methods (FM) technology makes a strong contribution to the verification required for safety-critical systems [15]. It is further recognized that FM will need to be integrated [3] in as “black-box” as possible a manner in order to achieve serious industry penetration. The B method of J.-R. Abrial [1, 19] is a formal method with good tool support [2, 8], and a good industrial track record, e.g. [9]. At Southampton, we have for some years been developing an approach of integrating formal specification and verification in B, with the UML [7]. The UML-B [22] is a profile of UML that defines a formal modelling notation combining UML and B. It is supported by the U2B tool [20], which translates UML-B models into B, for subsequent formal verification. This verification includes model-checking with the ProB model-checker [13] for B. These tools have all been developed at Southampton, and continue to be extended in current work.

Failure detection and management for engine control

A common functionality required of many systems is to detect and manage the failure of its inputs. This is particularly pertinent in aviation applications where lack of tolerance to failed system inputs could have severe consequences. The failure manager filters inputs from the controlled system, providing the best information possible and determining whether a transducer or system component has failed or not.

Inputs may be tested for magnitude, rate of change and consistency with other inputs. When a failure is detected it is managed in order to maintain a usable set of input values for the control subsystem and provide ‘graceful degradation’. To prevent over-reaction to isolated transient values, a failed condition must be confirmed as persistent before irreversible action is taken. Failure detection and management (FDM) in engine control systems is a demanding application area, see e.g. [6], giving rise to far more than a simple parameterizable product line situation.

Our approach contributes to the failure detection and management domain by presenting a method for the engineering, validation and verification of generic requirements for product-line engineering purposes. The approach exploits genericity both *within* as well as *between* target system variants. Although product-line engineering has been applied in engine and flight control systems [12, 10], we are not aware of any such work in the FDM domain. We define generic classes of failure-detection test for sensors and variables in the system environment, such as rate-of-change, limit, and multiple-redundant-sensor, which are simply instantiated by parameter. Multiple instances of these classes occur in any given system. Failure confirmation is then a generic abstraction over these test classes: it constitutes a configurable process of execution of specified tests over a number of system cycles, that will determine whether a failure of the component under test has occurred. Our approach is focussed on the genericity of this highly variable process.

Fault Tolerance

This application domain (and our approach to it) includes fault tolerant design in two senses: tolerance to faults in the environment, and in the control system itself. The FDM application is precisely about maximizing tolerance to faults in the sensed engine and airframe environment. The control system (including the FDM function) - is supported by a backup control system in a dynamically redundant design. This backup system - with distinct hardware/software design, with a reduced-functionality sensing fit - can be switched in by a watchdog mechanism if the main system has failed.

In the narrower (and more usual) sense, we will be examining various schemes for designing fault tolerance into the FDM software subsystem. Work to date has specified and validated a generic requirements specification for FDM. As we apply refinement techniques and technology to construct the design, we will consider various relevant approaches, such as driving the specification of a control system from environmental requirements [25], or the use of fault-tolerant patterns for B specifications [27] and their refinements [26].

Methodology

The process for obtaining a generic model of requirements is illustrated in Fig. 1. The first stage is an informal domain analysis which is based on prior experience of developing products for the application domain of failure detection and management in engine control. A taxonomy of the kind of generic requirements found in the application domain is developed and, from this, a *first-cut* generic entity-relationship model is formed by naming and relating the generic requirements.

The identification of a useful generic model is a difficult process warranting further exploration. This is done in the domain engineering stage where a more rigorous examination of the first-cut model is undertaken, using UML-B, U2B and ProB. The model is animated by creating typical instances of its generic requirement entities, to test when it is and is not consistent. This stage is model validation by animation, using the ProB and U2B tools, to show that it is capable of holding the kind of information that is found in the application domain. During this stage the relationships between the entities are likely to be adjusted as a better understanding of the domain is developed. This stage results in a *validated* generic model of requirements that can be instantiated for each new application.

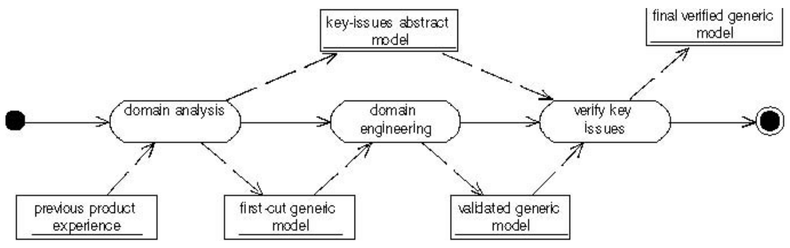


Fig. 1. Process for obtaining the generic model

For each new application instance, the requirements are expressed as instances of the relevant generic requirement entities and their relationships, in an *instance* model. The ProB model checker is then used to automatically verify that the application is consistent with the relationship constraints embodied in the generic model. This stage, producing a *consistent* instance model, shows that the requirements are a consistent set of requirements for the domain. It does not, however, show that they are the right set of requirements that will give the desired system behaviour.

Our aim in future work, therefore, is to add dynamic features to the instantiated model in the form of variables and operations that model the behaviour of the entities in the domain and to animate this behaviour so that the instantiated requirements can be validated. We would prefer to add this behaviour in the generic model so that it too can be re-used by the instantiated model.

During the domain analysis phase we found that considering the rationale for requirements revealed key issues, which are properties that an instantiated model should possess. Key issues are higher level requirements that could be expressed at a more abstract level from which the generic model is a refinement. The generic model could then be verified to satisfy the key issue properties by proof or model checking. This matter is considered in [21] which gives an example of refinement of UML-B models in the failure management domain.

The final stage is to validate the specific configuration. This would be done by providing actual values to generic behaviours when the generic mode is instantiated. The resulting specific model could then be animated to validate its behaviour.

Finally, we recognize the need for tools to support uploading of bulk system instance definition data, as well as the efficient and user-friendly validation/ debugging of said data. ProB could easily be enhanced to provide, for example, data counterexamples explaining invariant violations.

Domain Analysis

To obtain an initial understanding of the requirements domain we used domain analysis in a similar style to Lam [12]. The first step was to define the scope of the domain in discussion with engine controller experts. An early synthesis of the requirements and key issues were formed, giving due attention to the rationale for the requirements. Considering the requirements rationale is useful in reasoning about requirements in the domain [12]. For example, the rationale for confirming a failure before taking action is that the system should not be susceptible to spurious interference on its inputs. From the consideration of requirements rationale, key issues were identified which served as higher level properties required of the system. An example of such a property would be that the failure management system must not be held in a transient action state indefinitely. The rationale from which it has been derived is that a transient state is temporary and actions associated with this state may only be valid for a limited time.

A core set of requirements were identified from several representative failure management engine systems. For example, the identification of magnitude tests with variable limits and associated conditions established several magnitude test types; these types have been further subsumed into a general detection type. This type structure provided a taxonomy for classification of the requirements.

Domain analysis showed that failure management systems are characterised by a high degree of fairly simple similar units made complex by a large number of minor variations and interdependencies. The domain presents opportunities for a high degree of reuse within a single product as well as between products. For example, a magnitude test is usually required in a number of instances in a particular system. This is in contrast to the engine start domain addressed by Lam [12], where a single instance of each reusable function exists in a particular product. Our methodology is targeted at domains such as failure management where a few simple units are reused many times and a particular configuration depends on the relationships between the instances of these simple units. A first-cut entity relationship model was constructed from the units identified during the domain analysis stage. The entities identified during domain analysis were:

- **INP** Identification of an input to be tested.
- **COND** Condition under which a test is performed or an action is taken. (A predicate based on the values and/or failure states of other inputs).
- **DET** Detection of a failure state. A predicate that compares the value of an expression to be tested against a limit value.
- **CONF** Confirmation of a failure state. An iterative algorithm performed for each invocation of a detection, used to establish whether a detected failure state is genuine or transitory
- **ACT** Action taken either normally or in response to a failure, possibly subject to a condition. Assigns the value of an expression, which may involve inputs and/or other output values, to an output.
- **OUT** Identification of an output to be used by an action

Domain Engineering

The aim of the domain engineering stage is to explore, develop and validate the first-cut generic model of the requirements into a *validated* generic model. At this stage this is essentially an entity relationship model, omitting any dynamic features (except temporary ones added for validation purposes).

The first-cut model from the domain analysis stage was converted to the UML-B notation (Fig.2) by adding stereotypes and UML-B clauses (tagged values) as defined in the UML-B profile [22]. This allows the model to be converted into the B notation where validation and verification tools are available. The model contains *invariant* properties, which constrain the associations, and ensures that every instance is a member of its class. To validate the model we needed to be able to build up the instances it holds in steps. For this stage a constructor was added to each class so that the model could be populated with instances. The constructor was defined to set any associations belonging to that class according to values supplied as parameters.

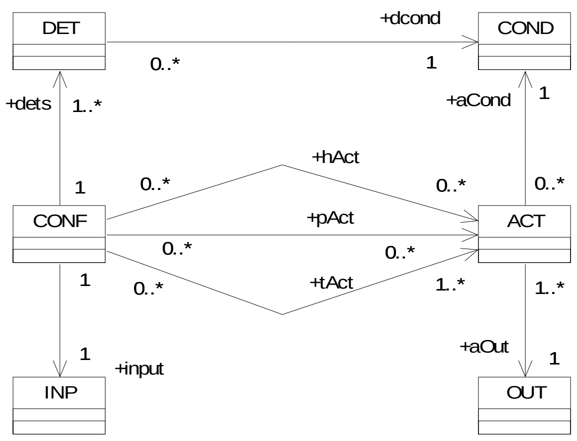


Fig. 2. Final UML-B version of generic model of failure management requirements

The model was tested by adding example instances using the animation facility of ProB and examining the values of the B variables representing the classes and associations in the model to see that they developed as expected. ProB provides an indicator to show when the invariant is violated. Due to the 'required' (i.e. multiplicity greater than 0) constraints in our model, the only way to populate it without violating the invariant would be to add instances of several classes simultaneously. However, we found that observing the invariant violations was a useful part of the feedback during validation of the model. Knowing that the model recognises inconsistent states, is just as important as knowing that it accepts consistent ones. The model was rearranged substantially during this phase as the animation revealed problems. Once we were satisfied that the model was suitable, we removed the constructor operations to simplify the corresponding B model for the next stage.

The next stage is to add behaviour to the generic model by giving the classes operations. In future work we will investigate the best way to introduce this behaviour during the process. It may be possible to add the behaviour after the static model has been validated as described above. Alternatively, perhaps the behaviour will affect the static structure and should be added earlier. In either case, we aim to formalise the rationale described in the domain analysis and derive the behaviour as a refinement from this.

Requirements for a specific application

Having arrived at a useful model we then use it to specify the requirements for a particular application by populating it with class instances. We use ProB to check the application is consistent with the properties expressed in the generic model. This verification is a similar process to the previous validation but the focus is on possible errors in the instantiation rather than in the model. The application is first described in tabular form. The generic model provides a template for the construction of the tables. Each class is represented by a separate table with properties for each entry in the table representing the associations owned by that class. The tabular form is useful as an accessible documentation of the application but is not directly useful for verification. To verify its consistency, the tabular form is translated into class instance enumerations and association initialisation clauses attached to the UML-B class model. This is done manually, which is tedious and error prone, but automation via a tool is envisaged.

ProB is then used to check which conjuncts of the invariant are violated. For our FDM example, several iterations were necessary to eliminate errors in the tables before the invariant was satisfied. Initially, testing of the instantiation caused an invariant violation. The ProB 'analyse invariant' facility provides information about which conjuncts of the invariant are violated. For example, a few conjuncts from the FDM example are shown:

```
(ACT:POW(ACT_SET)) == TRUE
(OUT:POW(OUT_SET)) == TRUE
(aOut:TotalSurjection(ACT,OUT)) == false
(aCond:(ACT-->COND)) == false
```

We found that the analyse invariant facility provided useful indication of where the invariant was violated (i.e. which conjunct) but, in a data intensive model such as this, it is still not easy to see which part of the data is at fault. It would be useful to show a data counterexample to the conjunct (analogous to an event sequence counterexample in model checking). This is another area for potential tool support.

Classification of problems

It would be useful to classify the kinds of problems found during animation and verification in order to better understand the source of problems and improve the requirements engineering process. So far, we have found that problems can be classified on a methodological stage basis. Possible categories on this basis, some of which we have experienced, are as follows.

- Verification of generic model – the generic model is inconsistent or incorrect
- Validation of generic model – the generic model is correct and consistent but does not reflect the generic requirements
- Validation of generic requirement – the generic model works as expected but animation leads expert to review generic requirements
- Verification of instantiation - the instantiation is inconsistent with the generic model because of an incorrect instantiation
- Verification of instantiation - the instantiation is inconsistent with the generic model because the generic model is inadequate
- Validation of instantiation - the instantiation is consistent with the generic model but does not reflect the specific requirements

- Validation of specific requirements - the instantiation is consistent with the generic model but animation leads expert to review specific requirements

In the future, when behavioural features are modelled, we expect to find other ways of classifying problems. For example we may be able to distinguish functional areas that are prone to incorrect specification.

Conclusion

In this paper we have discussed a product-line approach to the rigorous engineering, validation and verification of generic requirements for critical systems such as failure management and detection for engine control. The approach can be generalised to any relatively complex system component where repetitions of similar units indicate an opportunity for parameterised reuse but the extent of differences and interrelations between units makes this non-trivial to achieve. The product-line approach amortises the effort involved in formal validation and verification over many instance applications. So far we have considered the static, entity-relationship features of the requirements. In future work we aim to extend the approach to consider also the detailed meaning (i.e. dynamic behaviour) of these entities.

Two broad areas of future work are indicated by the case study, both linking to related work on Product Line Engineering (PLE). The first concerns instance data management, the second variability vs. commonality in the generic model.

For a product family such as FDM at ATEC as currently envisaged, instance data management is in principle straightforward. This is because no system instance/variant requirements are defined at the generic level – all structure and behaviour is specified in terms of a single generic model. Instance/variant requirements are captured completely by instance-level data. This means that all instance data structures are defined in terms of the generic class definitions. Therefore, the data for a system instance is simply defined as a subset of the database of all required instance specifications; tooling is thus a straightforward database application.

Instance management becomes more complex when variability is required in the generic model. This is the usual state of affairs in PLE. The mobile phone scenario of [16] is typical, where each system instance is defined by a distinct set of functional features, aimed at a specific market segment and target price. Features are not in general simply composable, and the totality of features cannot in general be specified in one generic model: variability specification is required in the generic model. To date approaches to this (such as [16]) have been in the obvious syntactic form: in ATEC for example, variants on the generic model for other engine manufacturers might be described as extra colour-coded classes, associations, states, events etc. A system variant (or sub-family) would thus be defined in terms of some colour-combination submodel. A more sophisticated metamodelling approach to variability specification, based on the Model-Driven Architecture of the OMG, has recently been proposed [17].

Future work will investigate developing such variability and tooling issues in the ATEC context, using the UML-B and refinement approaches already discussed. The application of refinement approaches to PLE to date has been modest, e.g. [5, 24], and has, in our view, much potential. Retrenchment, a generalizing theory for refinement, has been investigated in a feature engineering context [4], and may well also be useful in PLE.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial. http://www.atelierb.societe.com/index uk.html, 1998. Atelier-B.

[3] P. Amey. Dear sir, Yours faithfully: an everyday story of formality. In F. Redmill and T. Anderson, editors, *Proc. 12th Safety-Critical Systems Symposium*, pages 3–18, Birmingham, 2004. Springer.

[4] R. Banach, M. Poppleton. Retrenching Partial Requirements into System Definitions: A Simple Feature Interaction Case Study, 2003, *Requirements Engineering Journal* Vol. 8 (4)

[5] D. Batory, J.N. Sarvela, and A. Rauschmayer, Scaling Step-Wise Refinement, *IEEE Transactions on Software Engineering (IEEE TSE)*, June 2004

[6] C.M. Belcastro. Application of failure detection, identification, and accomodation methods for improved aircraft safety. In *Proc. American Control Conference*, volume 4, pages 2623–2624. IEEE, June 2001.

[7] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language -a Reference Manual*. Addison-Wesley, 1998.

[8] D. Cansell, J.-R. Abrial, et al. B4free. A set of tools for B development, from http://www.b4free.com, 2004.

[9] B. Dehbonei and F. Mejia. Formal development of safety-critical software systems in railway signalling. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, chapter 10, pages 227–252. Prentice-Hall, 1995.

[10] S.R. Faulk. Product-line requirements specification (PRS): an approach and case study. In *Proc. Fifth IEEE International Symposium on Requirements Engineering*. IEEE Comput. Soc, Aug. 2000.

[11] M. Fayad and D. Schmidt. Object-oriented application frameworks. *Communications of the ACM,* 40(10):32–38, Oct. 1997.

[12] W. Lam. Achieving requirements reuse: a domain-specific approach from avionics. *Journal of Systems and Software*, 38(3):197–209, Sept. 1997.

[13] M. Leuschel and M. Butler. ProB: a model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proc. FME2003: Formal Methods*, volume 2805 of *LNCS*, pages 855–874, Pisa, Italy, September 2003. Springer.

[14] R. Macala, L. Jr. Stuckey, and D. Gross. Managing domain-specific, product-line development. *IEEE Software*, pages 57–67, May 1996.

[15] UK Ministry of Defence. Def Stan 00-55: Requirements for safety related software in defence equipment, issue 2. http://www.dstan.mod.uk/data/00/055/02000200.pdf, 1997.

[16] D. Muthig. *GoPhone - A Software Product Line in the Mobile Phone Domain,* IESE-Report No. 025.04/E (Fraunhofer Institut Experimentelles Software Engineering, 2004

[17] D. Muthig and C. Atkinson. *Model-Driven Product Line Architectures,* In G.J. Chastel (Ed.): *Software Product Lines, Second International Conference, SPLC 2002, Proceedings*. LNCS 2379 Springer 2002, pages 110-129

[18] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Sofkvare Engineering,* SE-2, March 1976.

[19] S. Schneider. *The B-Method*. Palgrave Press, 2001.

[20] C. Snook and M. Butler. U2B -a tool for translating UML-B models into B. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[21] C. Snook, M. Butler, A. Edmunds, and I. Johnson. Rigorous development of reusable, domain-specific components, for complex applications. In J. Jurgens and R. France, editors, *Proc. 3rd Intl. Workshop on Critical Systems Development with UML,* pages 115–129, Lisbon, 2004.

[22] C. Snook, I. Oliver, and M. Butler. The UML-B profile for formal systems modelling in UML. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems*, chapter 5. Springer, 2004.

[23] W. Tracz. DSSA (Domain-Specific Software Architecture) pedagogical example. *ACM Software Engineering Notes*, pages 49–62, July 1995.

[24] A. Wasowski. Automatic generation of Program Families by Model Restrictions, In *R.L. Nord (Ed.): Software Product Lines, Third International Conference, SPLC 2004, Proceedings*. LNCS 3154 Springer 2004, pages 73—89

[25] I.J. Hayes, M. A. Jackson, and C. B. Jones, Determining the specification of a control system from that of its environment, In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proc. FME2003: Formal Methods*, volume 2805 of LNCS, pages 154–169, Pisa, Italy, September 2003. Springer.

[26] L. Laibinis and E. Troubitsyna, Refinement of fault tolerant control systems in B Source: Computer Safety, Reliability, and Security. 23rd International Conference, SAFECOMP 2004. Proceedings (Lecture Notes in Comput. Sci. Vol.3219), 2004, p 254-68

[27] L. Laibinis and E. Troubitsyna, Fault tolerance in a layered architecture: a general specification pattern in B, *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, 2004, p 346-55

Analyzing Fault-Tolerant Systems with FAUST

C. Ponsard, P. Massonet, and J.F. Molderez

CETIC Research Center, Charleroi (Belgium) - {cp,phm,jfm}@cetic.be

Abstract. Producing high quality requirements is the key to the successful design and development of the ever more complex systems controlling our present world. The KAOS goal-oriented requirements engineering methodology proved successful for this task, by enabling mixed semi-formal and formal modeling and reasoning about system properties at an early stage. This paper demonstrates the use of the FAUST toolbox to support the design of fault-tolerant systems based on this methodology.

1 Introduction

Our world is increasingly relying on complex software-based systems. In a growing number of fields such as transportation, finance, health-care, they now play a critical role as their failure can lead to catastrophic consequences in term of loss of company profit or even human lives. Hence they require high assurance for properties like security, safety, availability, etc.

Achieving assurance requires quality throughout the whole development lifecycle: from requirements to specification, architecture, code and tests. Among those, it is widely recognized that cause #1 of project failure still remains the poor quality of requirements. Our focus is on the requirements problem in relation with the rest of the lifecycle. More precisely, the scope of our work is to answer the following questions, depicted in figure 1.

– *Validation:* do we address the "right" requirements?
– *Verification:* are those requirements "right" ? Especially, in the scope of fault-tolerant systems: are those robust w.r.t. to what can go wrong ?
– *Acceptance:* is the deliverable right ? Can we test it with a good coverage w.r.t. to the wished/unwished properties ?

Our approach to address those questions is based on the elaboration of a goal model which captures the system and environment properties as well as the agent capabilities/responsibilities. This is well adapted for designing fault-tolerant systems because those should rely on a minimum set of well identified assumptions, and should react in a safe and graceful way when those are broken. The methodology also allows the analyst to reason in a pessimistic way about its model: starting from a simple and tractable model, the analyst can apply obstacle analysis to generate a number of obstacles to the wished properties. Based on risk assessment, those obstacles can then be eliminated/mitigated/tolerated to produce more robust requirements. This model has also two levels of description:

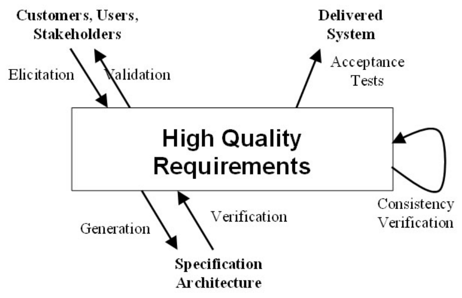


Fig. 1. Scope of the FAUST toolbox

1. *a semi-formal level* with graphical notation which integrates with standard UML notations (such as use cases, class, sequence diagrams). It is appropriate for acquiring the structure of the model and enough for non-critical properties.
2. *a formal level* using a real-time temporal logic which ensures the formal correctness of the model. It is only used on critical parts and is a natural extension of the semi-formal level.

The FAUST toolbox [13] supports the process of validation, verification and test case generation. Within the scope of this paper, we will more specifically show how it can support activities related to obstacle analysis at those stages. As concrete running example, we will analyze parts of the the London Ambulance System [1].

The rest of this paper is structured as follows. Section 2 will give a quick background on the KAOS requirements language used here with a stronger focus on how fault-tolerance is managed using obstacle analysis. Section 3 will describe how it is supported by FAUST. Section 4 will focus on integration issues, both at tool level and together with other methodologies like B.

2 Modeling Fault-tolerant Systems with KAOS

A KAOS requirements model is composed of four sub-models: (i) the central model is the *goal model* which captures and structures the assumed and required properties; (ii) the *object model* captures the relevant vocabulary to express the goals; (iii) the *agent model* takes care of assigning goal to agent in a realizable way; (iv) the *operation model* details, at state transitions level, the work an agent has to perform to reach the goals he is responsible for.

2.1 The Goal and Object Models

Although the process of building those 4 models is intertwined, the starting point is usually a number of key properties of the system to-be. Those are expressed using *goals* which are statements of intent about some system (existing or to-be) whose satisfaction in general requires the cooperation of some of the agents forming that system. *Agents* are active components, such as humans, devices, legacy software or software-to-be components, that play some role towards goal satisfaction. Some agents thus define the software whereas the others define its environment. Goals may refer to services to be provided (functional goals) or to the quality of service (non-functional goals). Goals are described informally in natural language (InformalDef) and are optionally formalized in a real-time temporal logic (FormalDef) [4][8][10]. Keywords such as *Achieve*, *Avoid*, *Maintain* are used to name goals according to the temporal behavior pattern they prescribe.

In our example, the goals relate to the correct processing of incidents by allocating and tracking ambulances. A key goal in the system is to achieve ambulance mobilization in time. It can be stated as follows:

Goal Achieve[AmbulanceMobilized]

InformalDef: For every responded call about an incident, an ambulance able to arrive at the incident scene within 11 minutes should be mobilized. The ambulance mobilization time should be less than 3 minutes.

FormalDef: $(\forall cl : Call, ic : Incident)\ Responded(cl) \wedge About(cl, inc)$
$\Rightarrow \Diamond_{\leq 3m}(\exists amb : Ambulance) Mobilized(a, inc) \wedge$
$\bullet\ [Available(amb) \wedge TimeDist(amb.loc, inc.loc) \leq 11m]$

In the above formulation, we have identified the *Call*, *Incident* and *Ambulance* entities with some of their attributes (such as *location* and *responded*) and relationships (*About* and *Mobilized*). Those are incrementally added to the structural model which captures passive (entities, relationships and events) and active objects (agents).

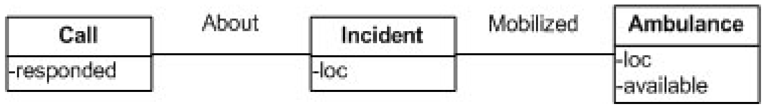


Fig. 2. Object Model

Unlike goals, *domain properties* are descriptive statements about the environment, such as physical laws, organizational norms or policies, etc. (eg. a crew member may forget to perform some required operation under stress).

A key characteristic of goal-oriented requirements engineering is that *goals* are structured and that guidance is provided to discover that structure and refine it until agents can be found to realize those goals in cooperation. In KAOS, *goals* are organized in AND/OR refinement-abstraction hierarchies where higher-level goals are in general strategic, coarse-grained and involve multiple agents whereas lower-level goals are in general technical, fine-grained and involve fewer agents [5]. In such structures, *AND-refinement* links relate a goal to a set of subgoals (called refinement) possibly conjoined with *domain properties*; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. *OR-refinement* links may relate a goal to a set of alternative refinements.

Figure 3 shows the goal structure for our system. It was set up starting from a few initial goals and by asking respectively "WHY" and "HOW" questions to

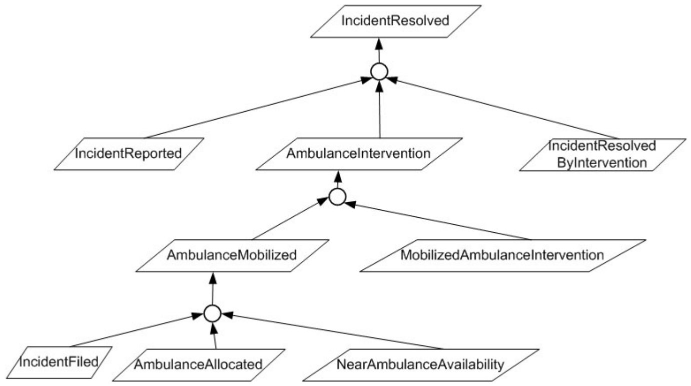


Fig. 3. Portion of the LAS goal graph showing AND-refinements

discover parent goals (such as *Achieve[AmbulanceIntervention]*) and son goals (such as *Achieve[AmbulanceAllocated]*).

2.2 The Agent Model

Goal refinement ends when every subgoal is realizable by some individual *agent* assigned to it, that is, expressible in terms of conditions that are monitorable and controllable by the agent [9]. A *requirement* is a terminal goal under responsibility of an agent in the software-to-be; an *expectation* is a terminal goal under responsibility of an agent in the environment. Agent are either human or automated (hardware or software).

The LAS system is a complex system with many interacting agents, both human (call reporter, call assistants, ambulance crew) and automated (AVLS: Automated Vehicle Location System, MDT: on board Mobile Data Terminal, CAD: Computer Aided Dispatch). Each agent can be described by his kind, his capabilities to monitor/control and the (realizable) goals under his responsibility. The AVLS can be described as follows:

Agent AVLS
Kind: Automated
ResponsibleOf: Maintain[AccurateAmbulanceLocation]
Monitors: *Ambulance.loc*
Controls: *AmbulanceInfo.loc*

The *agent interface view* displays the flow of monitored/controlled information among all agents and is a starting point for further architectural refinement using downstream methodologies, either traditional (like structured analysis [6]) or correct-by-construction (like B [2]).

Note it is also important to capture any assumption about agent behaviors, such as the possible deviations of human agents not complying with orders or the failure modes of hardware agents. Some of those may already be required for the initial (often overidealistic) goal-model and will be looked at systematically during the obstacle analysis.

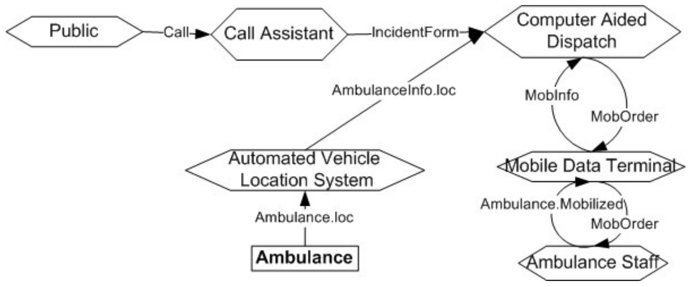


Fig. 4. Agent Interface Model

2.3 The Operation Model

Goals are operationalized into specifications of operations to achieve them [4]. An *operation* is an input-output relation over objects; operation applications define state transitions along the behaviors prescribed by the goal model. The specification of an operation is classical with precondition (necessary), postcondition (target state) and trigger condition (sufficient). An important distinction is also made between (descriptive) domain pre/postconditions and (prescriptive) pre-, post- and trigger conditions required for achieving some underlying goal(s). For example, the $Mobilize$ operation may be specified as follows:

Operation Mobilize
  Input: inc:Incident
  Output: amb:Ambulance, Mobilized
  DomPre: $\neg(\exists amb : Ambulance) Mobilized(amb, inc)$
  DomPost: $Mobilized(amb, inc)$
  ReqPre: for $AmbulanceMobilized$
    $Available(amb) \wedge TimeDist(amb.loc, inc.loc) \leq 11m$
  ReqTrig: for $AmbulanceMobilized$
    $(\exists cl : Call)\ Responded(cl) \wedge About(cl, inc)$

A goal operationalization is a set of such specifications.

2.4 Producing Robust Requirements

The correctness of all refinements in a goal model does not ensure that the specification is consistent: inconsistencies can occur between goals. First-sketch models also tend to be over-ideal and are likely to be violated from time to time in the running system due to the unexpected behavior of agents. The lack of anticipation of such behaviors may lead to unrealistic, unachievable and/or incomplete requirements. Such exceptional behaviors are captured by formal assertions called *obstacles* to goal satisfaction. Performing conflict and obstacles analysis is thus crucial for achieving high quality requirements[15] [16]. In this paper, we will only focus on obstacle analysis as conflict are not yet managed by FAUST.

Let G be a goal and Dom a set of domain properties. Following [16], an assertion O is said to be an obstacle to G in Dom iff the following conditions hold:

1. obstruction: $\{O, Dom\}| = G$
2. domain-consistency: $\{O, Dom\} \models false$
3. feasibility: there exists a scenario S producing a behavior H such that $H \models O$

Obstacles can be seen as the dual of goals. Like goals, obstacles can also be AND and OR refined with similar semantics to goal-refinement[16]. To discover obstacles, a systematic regression technique can be used. Starting from the negation of the goal, the procedure is to systematically regress through domain properties to look for possible causes (abduction). This process can be guided by a number of previously identified and classified obstacle refinement patterns. For example, taking the negation of the following goal:

Goal Achieve[MobilizedAmbulanceIntervention]
  UnderResponsibility AmbulanceCrew
  Refines AmbulanceIntervention
  FormalDef $(\forall a : Ambulance, inc : Incident)$
$$Mobilized(a, inc) \wedge TimeDist(a.loc, inc.loc) \leq 11m \Rightarrow \Diamond_{\leq 11m} Intervention(a, inc)$$

yields the following high level obstacle:

Obstacle MobilizedAmbulanceNotInTimeAtDestination
  FormalDef: $\Diamond(\exists a : Ambulance, inc : Incident)\ Mobilized(a, inc) \wedge$
$$TimeDist(a.loc, inc.loc) \leq 11m \wedge \Box_{\leq 11m} \neg Intervention(a, inc)$$

Looking at and obstacle refinement matching to the above obstacle, the following OR-refinement can be identified.

Obstacle AmbulanceMobilizationRetracted
  FormalDef: $\Diamond(\exists a : Ambulance, inc : Incident)\ Mobilized(a, inc) \wedge$
$$TimeDist(a.Loc, inc.Loc) \leq 11m \wedge (\neg Intervention(a, inc)\, \mathcal{U}_{\leq 11m} \neg Mobilized(a, inc))$$

Obstacle MobilizedAmbulanceStoppedOrInWrongDirection
  FormalDef: $\Diamond(\exists a : Ambulance, inc : Incident)\ Mobilized(a, inc) \wedge$
$$TimeDist(a.loc, inc.loc) \leq 11m \wedge$$
$$(\neg Intervention(a, inc)\, \mathcal{U}_{\leq 11m} TimeDist(a.loc, inc.loc) \leq TimeDist(\bullet a.loc, inc.loc))$$

Further refinement of these formal obstacles based on regression, patterns, and heuristics yield the following obstacle OR-refinement tree:

$\rightarrow MobilizedAmbulanceNotInTimeAtDestination$
  $\rightarrow AmbulanceMobilizationRetracted$
    $\rightarrow MobilizedAmbulanceDestinationChanged$
      $\rightarrow LocationConfusedByCrew$
  $\rightarrow MobilizedAmbulanceDestinationForgotten$
  $\rightarrow AmbulanceMobilizationCancelled$
$\rightarrow MobilizedAmbulanceStoppedOrInWrongDirection$
  $\rightarrow AmbulanceStopped$
  $\rightarrow AmbulanceBreakdownOrAccident$
  $\rightarrow AmbulanceStoppedInTraffic$

Those obstacles can then be resolved using one of the available strategies. For example, considering the obstacle $MobOrderTakenByOtherAmbulance$:

- *Obstacle prevention*, it could be possible to design the system to make it unfeasible by implementing the *Avoid*[*AmbulanceMobilizedWithoutOrder*] goal.
- *Obstacle mitigation*: the obstacle is not avoided but its possible consequences are mitigated. A way to avoid multiple ambulance mobilization and possible resource exhaustion is to implement $MobilizationByOtherAmbulanceKnown$, possibly using radio communications.
- *Obstacle reduction*: the obstacle is not prevented but measures are taken to reduce its occurrence. This could be achieved by a sector design.

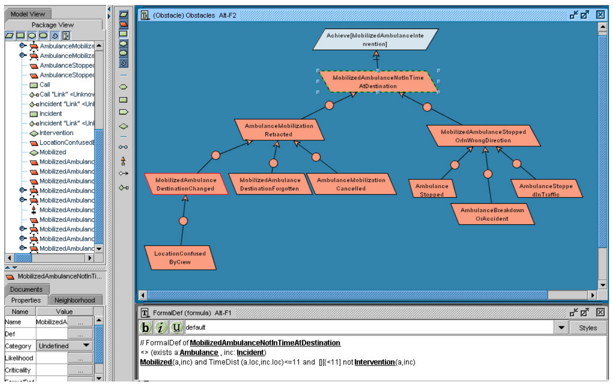
3 The FAUST toolbox in action

3.1 Verification using the Refinement Checker

The refinement checker can perform a variety of checks on the model in order to provide the formal assurance that goals are correctly refined, that operations enforce requirements, that obstacles are not present, etc. Through the use of model-checking, the checks are fully automated and produce suggestive counter-examples when they fail. The model-checker can also generate positive example of the negation of a goal which is an instance of obstacle. Figure 5 shows the tool in action on the formal obstacle refinements discussed in section 2.

3.2 Validation using the Requirements Animator

In order to validate the system requirements, the animator can simulate and display behaviors of the future system[14]. The simulation process relies on finite state machines (FSM) which are generated from a scoped subset of properties, enabling incremental validation. The user interface can display FSM in classical


(Obstacle) Obstacles Alt-F2
Achieve[MobilizedAmbulanceIntervention]
MobilizedAmbulanceNotInTimeAtDestination
AmbulanceMobilizationRetracted
MobilizedAmbulanceStoppedOrInWrongDirection
MobilizedAmbulanceDestinationChanged
MobilizedAmbulanceDestinationForgotten
AmbulanceMobilizationCancelled
AmbulanceStopped
AmbulanceBreakdownOrAccident
AmbulanceStoppedInTraffic
LocationConfusedByCrew
Model View
Package View
Documents
Properties
Neighborhood
FormalDef (formula) Alt-F1
default
Styles

```
// FormalDef of MobilizedAmbulanceNotInTimeAtDestination
<> (exists a:Ambulance , inc: Incident)
Mobilized(a,inc) and TimeDist (a.loc,inc.loc)<=11 and  [][(<11] not Intervention(a,inc)
```

Fig. 5. Checking Obstacle Refinements

state chart notations and with specifically designed graphical animations, based on domain notations familiar to the validating user. The animator can also be used for model debugging. For this purpose, a monitor automatically checks for the violation of all properties or occurrence of obstacles in the animation scope.

3.3 The Acceptance Test Generator

The acceptance test generator can produce a set of test cases from a non operational specification[11]. It is based on a goal coverage criterion which is appropriate for checking that a system matches its requirements. For now, only goal refinements are taken into account and mainly use the milestone and case-based refinement pattern for generating behavioral equivalence classes. The work is now being extended for generating dysfunctional test cases based on the information captured in the obstacle analysis process.

4 Integration issues

4.1 Tool Integration

The FAUST toolbox is centered on the model and each tool can benefit from the others. For example, counter-examples from the analyzer or tests cases from the test generator can be played into the animator. Figure 6 shows the interactions among the various components of the toolbox, those are mainly sub-models (goal or operational), various kind of traces and FSM. Some results are also interesting to export outside the toolbox for later use in other stages: test cases, FSM for code generation, etc.

The toolbox architecture is open and modular. It is currently available as extension of the Objectiver requirements platform [12] which provides a full (meta)conceptual repository with queries, checks, support for textual documents and graphical models, trace management, navigation, and a powerful document generator with templates for producing standard requirements documents.

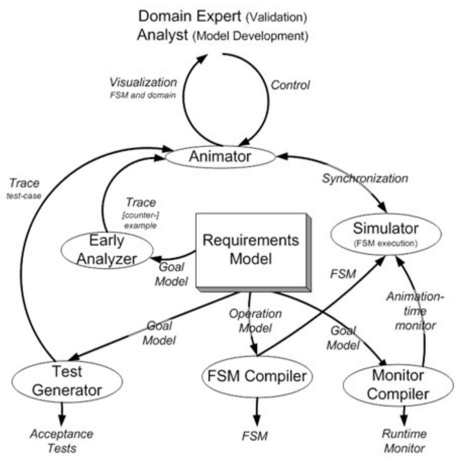


Fig. 6. Interactions among the FAUST tools

4.2 Method Integration - the B Connector

As the FAUST scope is located early in the development lifecycle, it is vital to to provide connectors to existing industrial development methods. We believe our approach is interesting because it provides the often missing link with the requirements and it enables the move of some parts of the formal reasoning a step ahead.

The method we are currently investigating is B [2], an industrial-strength formal method which is more and more focusing on system engineering (B-System evolution). This evolution is making it closer to our scope and goes in the direction of bridging the requirements gap [3]. For "top-down" engineering, it will help the analyst to identify key properties, detect design problems at an easy stage and produce B specifications which will be easier to refine and prove. For "bottom-up" re-engineering, it helps explaining the design to customers and managers, especially for showing that all requirements have been covered.

Within B-system, we are more specifically studying the CompoSys approach developed by ClearSy [7] which focuses on the modeling and integration of components in an industrial context (such as automotive and railway transportation systems). So far, the KAOS agent interface diagram was identified as fitting the level of description addressed by Composys and some mapping mechanisms are now being investigated on a practical case study.

5 Conclusions

To sum up, the benefits of the approach are:

- **Goal and agent based approach:** KAOS allows the analyst to capture, refine and reason about system properties within its environment, assigning responsibilities, exploring and comparing alternative designs.
- **Model-based approach** with automatic derivation of a wide variety of artefacts like semi-fomal documents and formal specifications (such as B),

acceptance test cases etc. It enables the easy integration in any lifecycle (whether test-based or correct-by-construction)

- **Access to the power of formal methods while preserving communication**: because formal notations and underlying tools can be hidden and explained in natural/graphical languages.
- **Reduced costs** because problems are detected and addressed earlier and the model can be used in connection with the rest of the development.

The toolbox is currently used internally on ongoing cases to help assess its limits and to drive the discovery of missing features. The current priority is to open the toolbox to further use later in the development steps with a focus on the B method as used in practice.

Acknowledgement

This work is financially supported by the European Union (ERDF and ESF) and the Walloon Region (DGTRE).

References

1. *Report of the inquiry into the london ambulance service*, The Communications Directorate, South West Thames Regional Authority, 1993.
2. J. R. Abrial, *The b-book: Assigning programs to meanings*, Cambridge University Press, 1996.
3. J.-R. Abrial, *B: passe, present, futur*, 2002.
4. A. Dardenne, A. van Lamsweerde, and Stephen Fickas, *Goal-directed requirements acquisition*, Science of Computer Programming **20** (1993), no. 1-2, 3–50.
5. R. Darimont and A. van Lamsweerde, *Formal refinement patterns for goal-driven requirements elaboration*, 4th FSE ACM Symposium, San Francisco, 1996.
6. T. Demarco, *Structured analysis and system specification*, Yourdon Inc, 1979.
7. G.Pouzancre and J.-P. Pitzalis, *Modelisation en b evenementiel des fonctions mecaniques, electriques et informatiques dun vehicule*, RSTI-TSI (2003).
8. R. Koymans, *Specifying message passing and time-critical systems with temporal logic, lncs 651*, Springer-Verlag, 1992.
9. E. Letier and A. van Lamsweerde, *Agent-based tactics for goal-oriented requirements elaboration*, 2002.
10. Z. Manna and A. Pnueli, *The reactive behavior of reactive and concurrent system*, Springer-Verlag, 1992.
11. J.F. Molderez and C. Ponsard, *Deriving acceptance tests from goal requirements*, 2nd International Mozart/Oz Conference, Charleroi (Belgium), September 2004.
12. The Objectiver Tool, *http://www.objectiver.com*.
13. The FAUST toolbox, *http://faust.cetic.be*, 2004.
14. H. Tran Van, A. van Lamsweerde, P. Massonet, and C. Ponsard, *Goal-oriented requirements animation*, 12th IEEE Int.Req.Eng.Conf., Kyoto, September 2004.
15. A. van Lamsweerde, R. Darimont, and E. Letier, *Managing conflicts in goal-driven requirements engineering*, IEEE Transactions on Software Engineering (1998).
16. A. van Lamsweerde and E. Letier, *Handling obstacles in goal-oriented requirements engineering*, IEEE Transactions on Software Engineering, Special Issue on Exception Handling **26** (2000), no. 10.

Rigorous Fault Tolerance Using Aspects and Formal Methods

Shmuel Katz
Computer Science Department
The Technion
Haifa, Israel

Abstract: The use of aspect-oriented software development (AOSD) and formal verification and analysis of aspects is suggested as a modular approach to adding fault tolerance to systems, under a variety of fault models. The properties of aspects are shown appropriate for such a modularization, and several example aspects for fault tolerance are described. Among these are aspects to treat self-stabilization, aspects to overcome crash failures, and aspects to overcome faulty communication. Some approaches to verification of aspects are also described and shown relevant for verifying fault-tolerance after appropriate aspects are added to a non-fault-tolerant system.

Relevant activities of the EU network of excellence AOSD-Europe are outlined, in the framework of the four virtual labs of that network. Those labs deal with the areas of programming languages for aspects, requirements analysis and design, applications of aspects for middleware, and semantics and verification for systems with aspects. The intentions of the network to develop joint tools to aid in verification of systems with aspects are explained, as part of the research plan of the Formal Methods Laboratory of AOSD-Europe. Potential points for cooperation with RODIN are also suggested.

The Fault-Tolerant Insulin Pump Therapy

Alfredo Capozucca, Nicolas Guelfi and Patrizio Pelliccione

Software Engineering Competence Center
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
Luxembourg, L-1359 -Luxembourg

Abstract. In this paper we describe our experience using Coordinated Atomic Actions (CAAs) to design a control system for a medical treatment, which has high reliability requirements. The "Fault-Tolerant Insulin Pump Therapy" is based on the Continuous Subcutaneous Insulin Injection technique involving different sensors and actuators in order to enable continued execution of the treatment, as well as detect faults in it. Precisely that is the challenge raised by this example, to design a control system that maintains the delivery of insulin even in the presence of a large number and variety of hardware and software failures. The implementation of this control system has been made in Java using an extension of the DRIP framework, that ensures the reliability properties of systems designed using CAAs.

1 Introduction

Software and hardware systems have become increasingly used in many sectors, such as manufacturing, aerospace, transportation, communication, energy and healthcare. Failures due to software or hardware malfunctions and to malicious intentions can have economic consequences, but can also endanger human life. In fact, if a health care system breaks down, the effect on the hospital and patients could be huge. Therefore health care systems must be available 24 hours a day, seven days a week with no exceptions (*availability*).

Different approaches have been proposed in the literature to model medical systems. The Asynchronous Transfer Mode (ATM) network provides a robust and resilient network that is able to combine high performance with the Quality of Service (QoS), which are required by advanced and mission-critical telemedicine, and clinical applications [4, 6]. *Resilience* is the ability of systems to undergo abnormal situations without loss of its essential functions. A resilient system persists for a long time despite disturbances. More precisely, resilient systems should be able to ensure their services even when some system parts have abnormal behaviors due to degradation of the components, unavailability or attack.

In this paper we focus on Coordinated Atomic Actions (CAAs) as a design structuring concept to ensure the needed requirements of reliability and availability [9] and on the framework called Dependable Remote Interacting Processes (DRIP) [10] that embodies CAAs in terms of a set of Java classes. Although the

DRIP framework supports the complete semantics of CAA, we had to change the implementation to fix some problems. Fundamentally, we have changed the notification of an exception from a composed CAA to the enclosed context, as well as the way in that each handler must be defined and linked with its corresponding manager and the internal mechanism to execute the handlers when an exception has to be handled. The change on the exception handling mechanism does not allow us to have nested handlers any more, that is a requirement for DRIP (but not for CAAs). Thus, this new framework only supports CAAs requirements. We refer to this new framework with the name CAA-DRIP and the full implementation details about it can be found in [2].

The aim of this paper is to show how CAAs and CAA-DRIP can be successfully used for medical systems which require resilience and availability. The case study that we consider concerns a diabetes control system that is aiming at correctly delivering the insulin on a patient. The doctor suitably sets the parameters and miniaturized sensors and pumps check the patient status and administer the insulin. It is of primary importance, for the patient health, that the whole application works properly 24 hours a day without interruption.

In Section 2 we give an introduction to CAA and CAA-DRIP, and in Section 3 we show the design and the implementation of the considered case study . The paper closes with conclusions and future works.

2 Background

In this section we introduce CAAs and the requirements of the CAA-DRIP framework that are used in the following of the paper.

2.1 Coordinated Atomic Actions

Backward error recovery (based on rolling system components back to the previous correct state) and forward error recovery (which involves transforming the system components into any correct state) represent the two main approaches for error recovery. The former uses either diversely-implemented software or simple retry; the latter is usually application-specific and relies on exception handling mechanisms. Distributed transactions [3] are a well-known technique that uses backward error recovery as the main fault tolerance measure in order to satisfy completely or partially the ACID (atomicity, consistency, isolation, durability) properties. Atomic actions [1] allow programmers to apply both backward and forward error recovery.

CAAs have been proposed by Xu et al. [7] in order to combine distributed transactions and atomic actions assuring consistent access to objects in the presence of concurrency and potential faults. If an exception is raised into a CAA, then an exception handler tries to recover them. In the positive case the CAA terminates normally, on the contrary, it attempts to roll-back the state of external objects. Finally, an unsuccessful roll-back causes a failure. CAAs can be nested and in this case exceptions raised by a nested CAA are propagated to the enclosing one.

2.2 CAA-DRIP

The CAA Dependable Remote Interacting Processes (CAA-DRIP) framework comprises a set of Java classes supporting CAAs.

CAA-DRIP relies on the notion of Dependable Multiparty Interaction (DMI) [8]. The main properties of a multiparty interaction are (i) using a guard to check the preconditions to execute the interaction, hence (ii) the need for having synchronization upon entry of participants; (iii) using an assertion, after that the interaction has finished, to check that a set of post-conditions has been satisfied by the execution of the interaction; (iv) and, finally, atomicity of external data to ensure that intermediate results are not passed to the outside processes before that the interaction finishes. These properties make DMIs an excellent vehicle for implementing reliable applications. Zorzo and Stroud have proposed within CAA-DRIP a general scheme for designing DMIs in a distributed object-oriented system [10]. DMIs extend the notion of multiparty interaction to include facilities for handling exceptions, which allows dealing with failures in one or more participants of the multiparty interaction, and in particular concurrent exceptions and synchronization upon exit.

CAAs can be derived from DMIs by adopting a more restricted form of exception handling with a stronger exception handling semantics. CAA-DRIP is designed to support this derivation and thus can be used to implement CAAs.

3 The Diabetes Control System

The Diabetes Control System makes use of different kinds of devices, which combine high performance, lower power consumption, and wireless communication, increasing the "intelligence" of medical sensors and actuators.

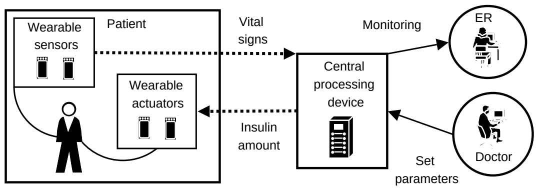


Fig. 1. The actors and their relationships.

Figure 1 shows the different actors present in our scenario. The main actor is the *patient* who is receiving the treatment and who has put on the wearable devices (sensors and actuators). The *doctor* must set the parameters for the devices to allow them to work according to the specific treatment that the patient has to receive. This information will be stored in the patient's personal record and will be consulted by the application. Moreover, the facilities for the doctor to change and consult the information about the treatment should be designed to be fault-tolerant, as well.

The last actor is the *emergency room (ER)*, where caregivers are continually monitoring the patient's vital signs. They will be the first to know if there is a problem with the treatment that the patient is receiving. The dotted arrows represent wireless connection and show how these wearable devices are connected to the central processing device. In our representation, the doctor and the ER are connected to the network in the traditional way, but they could also be connected to the network of the hospital by wireless connection. This paper focusses on the application that controls insulin delivery to the patient.

3.1 Requirements

The fault-tolerant diabetes control system will be used to implement the technique called Continuous Subcutaneous Insulin Injection [5]. This technique uses miniaturized sensors and pumps to check the patient's status and to administrate insulin, respectively.

In this scenario, two sensors are used: one to monitor the *current blood glucose level (CBGL)* and another one to check the *heart rate (HR)*. There are also two pumps: one for injecting *long acting insulin (LAIP)* and another one for injecting *rapid acting insulin (RAIP)*.

The patient's vital signs collected by the sensors are wirelessly sent to the central processing device, which is connected to the network of the hospital. These values are used to determine the patient's status and, fundamentally, to define the *basal rate* (the amount of insulin to inject) for each pump. This is calculated by a formula that takes into account the Target Blood Glucose Level (TBGL), the Duration of Insulin Action (DIA) according to the kind of insulin used and the patient's current values measured by the sensors (CBGL and HR). The TBGL and DIA parameters must be determined by the doctor, as well as the low limit of the cartridge of each pump and the safe insulin delivery limit. These values must be defined before the treatment is launched, but if it is necessary, these values can be changed while the patient is receiving the dose, as well. More details about elements included into this control process are given in the Appendix.

The result given by the formulas represents the amount of insulin that each pump must inject to keep the blood glucose as near as possible from the patient's target blood glucose. In this way, the central processing device commands each actuator to inject the corresponding amount of insulin. The insulin will then arrive to the patient by a cannula, which is a small soft tube, inserted into the patient's body. Each actuator has also a sensor, which provides useful information about it. The application, with the help of these sensors gets information on the current status of each actuator.

When an error occurs, it must be detected and, depending on the seriousness, the control program either tries to solve it (first attempting the operation or, in second place, using the values of the previous cycle), or the control system turns on the alarm and stops the delivery of insulin. The ER personnel detects the alarm, solves the problem and then turns off the alarm.

3.2 Design

The functional requirements presented in the previous section drive the application design through the definition of seven CAAs (Figure 2). These CAAs were designed using **nesting** and **composing**. Nesting is defined as a subset of the roles (*Params* and *Controller*) of a CAA (**CAA_Cycle**) defining a new CAA (**CAA_Checking/CAA_Executing**) inside the enclosing CAA (**CAA_Cycle**).

CAA_Cycle works like a container for the nested **CAA_Checking** and **CAA_Executing** CAAs. Its main task is to determine the amount of insulin that must be injected for each pump. These amounts of insulin are defined by the *InsulinAmount* algorithm, which is used by the *Calculus* role.

CAA_Checking and **CAA_Executing** CAAs were defined to isolate the execution of a group of tasks to determine the insulin amount as well as to deliver them on the respective pumps. **CAA_Checking** provides the input information for the algorithm used by *Calculus* and its output is passed to **CAA_Executing** which uses this information to deliver the insulin.

The "interactions" between roles are represented in Figure 2 by vertical wide solid arrows (not to be confuse with roles which are represented by horizontal thin solid arrows).

CAA_Checking has to retrieve the parameters set by the doctor for the patient and also, it has to get the values of the sensors. **CAA_Executing** sends commands to each pump and registers in a log the original commanded values and those that were really injected. Both nested CAAs use *Controller* and *Params* roles to achieve their goals (which have been explained before). The fact that the roles are embedded in two different nested CAAs, allows to hide the tasks that the roles do for the first CAA with respect to the second one.

The log is an external object, and the access to it is represented by wide slashed arrows. The patient, his personal record and the pumps are external objects, as well.

We defined four more CAAs to perform the activities corresponding to the sensors and actuators. The first one is **CAA_Sensors**, which is in direct contact with the wearable devices that have to get the patient's vital sign. The second one is **CAA_Actuators**. Both CAAs are composed.

Composed CAAs are different from the nested in the sense that the first one is an autonomous entity with its own roles and external objects. The internal structure of a composed CAA (e.g. **CAA_Sensors**), i.e. set of roles (*S_CT*, *BGC* and *HR*), accessed external objects (*Patient* and *Patient's record*) and behavior of roles, is hidden from the calling CAA (**CAA_Checking**). The *Controller* role that calls the composed **CAA_Sensors** synchronously waits for the outcome. Then, the calling role resumes its execution according to the outcome of the composed **CAA_Sensors**. If the composed **CAA_Sensors** terminates exceptionally, its calling role (which belongs to **CAA_Checking**) raises an internal exception which is, if possible, locally handled. If local handling is not possible, the exception is propagated to all the peer roles of **CAA_Checking** for coordinated error recovery.

CAA_Actuators contains the composed **CAA_RAIP** and **CAA_LAIP** CAAs. Each composed CAA manages both a pump and a sensor. This sensor allows us to know the state of the pump before and after the insulin injection.

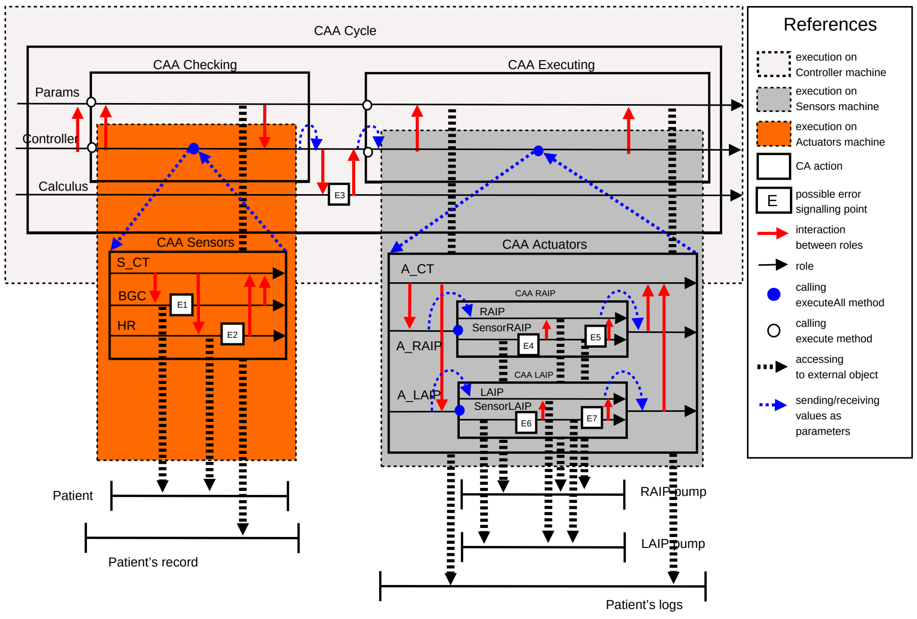


Fig. 2. The Design by CAAs and the faults that we are handling.

In **CAA_Sensors** and **CAA_Actuators** there is a special role (*S_CT* and *A_CT* respectively), who is in charge of data exchange among the roles that compose each CAA. Thus, this role receives/sends all the information from/to the enclosed/nested context. This data manipulation is done in the same way by the *RAIP* and *LAIP* roles. This way to send or receive information as parameters is represented by thin dotted arrows. As showed in Figure 2, the *Controller* machine receives information from the *Sensors* machine that commands the pumps that are running on the *Actuators* machine.

Failure definitions and analysis Before defining and analysing the different possible failures that may happen in our example, we have to state the assumptions that we have done: (i) The values of the sensors and of the actuator are always transmitted correctly, without any loss or error. (ii) Each failure on any sensor or actuator is indicated by a specific value, which shows which kind of failure happened. (iii) The alarm signalling mechanism is free of faults and does not fail.

Now, we can define and analyse various failures with respect to some elements that compose our scenario, as well as the basic requirements for handlers related to each exception that will be launched when an error is detected.

1. Sensor stops (E1 or E2): a wearable sensor could not send valid values. This failure is indicated automatically by a special value of the wearable sensor. The control system will try again getting the value to continue the cycle, but if

the problem persists the delivery will stop and the danger alarm will be turned on.
2. Delivery Limit (E3): there is an amount of insulin that should be delivered to keep the patient's target blood glucose which is dropping out of the safe range. In this case, the delivery is stopped and the danger alarm is turned on.
3. Actuator stops (E4, E6): a sensor that is monitoring an actuator has detected a problem before trying to inject the insulin. This means that the actuator is not properly working. In this case, the control program must stop the delivery of insulin and start to ring the danger alarm.
4. Delivery stops (E5, E7): a sensor that is monitoring an actuator has detected a problem after the insulin injection. It means that the actuator could not inject the required amount. The control program will try again to deliver the insulin, but if the problem goes on, the delivery of insulin will be stopped and the danger alarm will be turned on.
5. Cartridge very low (E4, E5, E6 or E7): the quantity of insulin in a cartridge is less than the low limit set in the cartridge. The basal delivery continues, but the warning alarm is turned on.
6. Cartridge empty (E4, E5, E6 or E7): a cartridge of a pump does not have any more insulin, thus the systems will be stopped and the danger alarm is turned on.

3.3 Implementation

This section describes the most important changes we made on DRIP [10] and how we used this new framework to implement our design. Due to space limitations, we just show the implementation of **CAA_Sensors** and how it is launched. Using this example, we give some ideas on the extensions made on DRIP. For more details, interested reader can refer to [2]. This CAA is composed by three roles and for each one of them we define a *Manager* (lines 2-4). Once the instantiation of these objects is done, we are able to define each *Role* object (lines 7-9) by instantiating a new class, which inherits from the *Role* class provided by the framework. We must give the name of the role, its manager and the leader manager each time that we define a new *Role* object. In this case, *mgrCT* is the leader manager and it is the responsible for the coordination of the each role when they must be executed, as well as, when an exception is raised.

Definition of CAA_Sensors

```
1  //Managers
2  mgrCT = new ManagerImpl("mgrCT","CAA_Sensors");
3  mgrCBGC = new ManagerImpl("mgrCBGC","CAA_Sensors");
4  mgrHR = new ManagerImpl("mgrHR","CAA_Sensors");
5
6  //Roles
7  roleCT = new CT("roleCT",mgrCT,mgrCT);
8  roleCBGC = new CBGC("roleCBGC",mgrCBGC,mgrCT);
9  roleHR = new HR("roleHR",mgrHR,mgrCT);
10
11 //Handlers for SensorStops exception
12 hndrSS_CT = new SensorStopsCT("hndrSS_CT",mgrCT,mgrCT);
13 hndrSS_CBGC = new SensorStopsCBGC("hndrSS_CBGC",mgrCBGC,mgrCT);
14 hndrSS_HR = new SensorStopsHR("hndrSS_HR",mgrHR,mgrCT);
15
```

```
16 //Binding between the Exception and the Handlers
17  Hashtable ehCT = new Hashtable();
18 ehCT.put(SensorStops.class ,hndrSS_CT);
19  Hashtable ehCBGC = new Hashtable();
20 ehCBGC.put(SensorStops.class ,hndrSS_CBGC);
21  Hashtable ehHR = new Hashtable();
22 ehHR.put(SensorStops.class ,hndrSS_HR);
23
24 //Setting the binding on each Manager
25 mgrCT.setExceptionAndHandlerList(ehCT);
26 mgrCBGC.setExceptionAndHandlerList(ehCBGC);
27 mgrHR.setExceptionAndHandlerList(ehHR);
```

If there is a problem in the normal execution, we have the chance to define an alternative behaviour. The lines 11-27 show how we can define this exceptional behavior. If these lines are not present, when an exception is raised, the CAA is stopped and the problem is forwarded to the enclosed context.

The lines 12-14 correspond to the definition of the handlers that are only executed when the exception *SensorsStops* is raised. On Figure 2 the errors E1 and E2 represent the places where this exception could happen. Each handler object defined is an instance of a new class derived from *Handler* class, which belongs to the framework. The class *Handler* has been introduced in CAA-DRIP to correctly manage the information context of the CAA where the exception has been raised [2]. For each exception that we want to handle in the CAA, we have to define n handlers, where n is the number of roles defined in the CAA. Each handler must be informed of its name, its manager (which must be one of the used in the definition of the roles) and the leader manager (not necessary the same used for the roles).

The next step is the explicit definition of the binding between the considered exception, and the handlers that have been defined to manage it. Each binding is represented by a hashtable, which is controlled by a manager (lines 17-22). Each manager (e.g. *mgrCT*) coordinates the execution of a role (e.g. *roleCT*). The role represents the normal behavior. In the case in which an exception is launched (*SensorStops*), each manager stops the execution of its associated role it starts to execute its associated handler (e.g. *hndrSS_CT*). Finally, we must set each hashtable on the corresponding handler that is managing each role and handler (lines 25-27).

The composed **CAA_Sensors** is launched from the *Controller* role. The definition of a role implies the *Role* class extension, which belongs to the framework and reimplement its *body* method. Inside this method we define the tasks that must be executed to achieve the requirements of the considered role. The following Java source code corresponds to the role *Controller* and shows how **CAA_Sensors** is called, as well as the interaction with the *Params* role and with **CAA_Sensors** CAA is achieved.

Launching CAA_Sensors

```

 public void body(Object list[]) throws Exception, RemoteException {
  try{
          //launching the Composed CAA_Sensors
          roleCT.executeAll(list);

```

```
7            //getting Composed CAA_Sensors outcomes
8            RemoteQueue rqOut = (RemoteQueue) list[0];
9            Integer bgcValue = (Integer)rqOut.get();
10           Integer hrValue = (Integer)rqOut.get();
11
12           //getting values from Params role
13           RecordPatient rp = (RecordPatient)paramsPatientQueue.get();
14
15           //passing information to CAA_Cycle
16           rqOut.put(bgcValue);
17           rqOut.put(hrValue);
18           rqOut.put(rp);
19
20    } catch (Exception e) {
21           //Local handling for Checking.Controller exception;
22           throw e;
23   }
```

The *body* method receives a list of objects as parameter (line 2), which is used to exchange information with its context. The *executeAll* method is used for the *roleCT* object (there is no difference about which CAA role is used) to launch the composed **CAA_Sensors** (line 5). This method takes an object list as parameter that is used to get the **CAA_Sensors** outcomes. Lines 8-10 show how we retrieve these outcomes from the list. Interaction among roles appears in line 13 and it represents an information flow from *Params* to *Controller*. Once the *Controller* role has all the patient's information it must send this information to the enclosing **CAA_Checking** (lines 16-18). If along the execution of these tasks an exception is raised, we have the chance to handler it locally inside the *catch* block. In this example, if an exception happens, it is directly passed to the enclosing context (line 22).

4 Conclusions and Future Work

In this experience paper we introduced a control system for a fault-tolerant insulin pump therapy. In order to ensure the needed requirements of reliability and availability, the system has been designed using the CAAs mechanism that offers approaches for error recovery. The implementation of the control system has been made in Java, using a variant of the DRIP framework, because along the implementation of this case study we found some problems in the original DRIP. These problems were fixed in a new framework which just supports CAA requirements and is called CAA-DRIP. On the future work side we plan to release CAA-DRIP explaining details on changes made.

Acknowledgments This work has benefited from a funding by the Luxembourg Ministry of Higher Education and Research under the project number MEN/IST/04/04. The authors gratefully acknowledge help from A. Romanovsky, P. Periorellis, R. Razavi and A. Zorzo.

References

1. R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering. IEEE Press*, pages pp. 811–826, 1986.

2. Correct Web Page. http://se2c.uni.lu/tiki/tiki-index.php?page=correctdoc, 2005.
3. J. Gray and A. Reuter. Transaction processing: Concepts and techniques. *The Morgan Kaufmann series in data management*, pages pp. 36–37, 1993.
4. P. Jain, S. Widoff, and D. C. Schmidt. The design and performance of medjava. *IEE/BCS Distributed Systems Engineering Journal*, December 1998.
5. National Institute for Clinical Excellence. Guidance on the use of continuous subcutaneous insulin infusion for diabetes. *http://www.nice.org.uk/pdf/57_insulin_pumps_fullguidance.pdf*. 2003.
6. G. Weiss. Welcome to the (almost) digital hospital. *IEEE Spectrum Online, http://www.ieeta.pt/sias/courses/imt/Resources/IEEE_AlmostDigitalHospital_Mar2002.pdf*, 2002.
7. J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.
8. A. Zorzo. Multiparty interactions in dependable distributed systems. *PhD Thesis, University of Newcastle upon Tyne, Newcastle upon Tyne, UK*, 1999.
9. A. Zorzo, A. Romanovsky, B. R. J. Xu, R. Stroud, and I. Welch. Using co-ordinated atomic actions to design complex safety-critical systems: The production cell case study. *Software-Practice and Experience*, pages pp. 667–697, 1999.
10. A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 435–446. ACM Press, 1999.

Appendix: "Terminology"

Basal rate: the amount of insulin delivered over 24 hours per day, providing a background of insulin at all times. The rate programmed is intended to keep the blood glucose within the user's Target Range (TR) between meals and overnight. The basal rate is measured in units per hour (u/hr).

Blood Glucose Level (BGL): the amount of glucose in the blood. BG levels average 100 mg/dl (5.5 mmol/L) for someone without diabetes. The healthcare provider help in determining the "target range" for the blood glucose level.

Heart Rate (HR): is the number of contractions of the heart in one minute. It is measured in beats per minute (bpm). When resting, the adult human heart beats at about 70 bpm (males) and 75 bpm (females), but this rate varies between people.

Duration of insulin action (DIA): a certain amount of time insulin is active and available in the body after it has been given by a subcutaneous bolus. Talking with the healthcare provider helps in determining the duration of the insulin action through blood glucose testing.

InsulinAmount algorithm: it is used to calculate the needed amount of insulin. The formula takes into account the Target Blood Glucose Level (TBGL), the Duration of Insulin Action (DIA) according to the type of insulin used, the current blood glucose (CBGL) and the current heart rate (HR). The result represents the needed amount of insulin.

$$InsulinAmount(TBGL, DIA, CBGL, HR)$$

Omnibus: A clean language and supporting tool for integrating different assertion-based verification techniques

Thomas Wilson, Savi Maharaj, Robert G. Clark

Department of Computing Science and Mathematics, University of Stirling, Stirling, Scotland
`{twi,sma,rgc}@cs.stir.ac.uk`

Abstract. Omnibus is a new system for the development of reliable Object-Oriented software. It includes a clean language that is superficially similar to Java but removes aspects that particularly complicate verification. Integrated support is provided for run-time assertion checking, extended static checking and full formal verification. The language is supported by a prototype IDE with a type checker, Java code generator, HTML documentation generator and a range of verifiers. This paper presents the case for Omnibus, gives an overview of the language and tools and discusses its relationship to dependable systems development.

1. Introduction

There are three distinct assertion-based approaches for the integrated specification, implementation and verification of Object-Oriented (OO) software: run-time assertion checking [14], extended static checking [10] and full formal verification [4]. Full formal verification offers the possibility of producing error-free software whereas run-time assertion checking and extended static checking have more modest aims, attempting to catch only a subset of assertion violations. Most existing tools are built around a single one of these approaches. However, the approaches have complementary strengths. Full formal verification is ideal for supporting reusable software components and verifying critical modules within a system though its cost cannot typically be justified for systems in their entirety. Run-time assertion checking and extended static checking offer a better compromise for the majority of application-specific code. They require more modest additional investments but are inadequate for writing critical code and reusable components. We propose the use of these approaches together within different parts of a single system, using full formal verification for reusable components and critical modules and a combination of run-time assertion checking and extended static checking for the remainder of the code.

The starting point for an assertion-based approach is the language, of which there are two kinds: ones that extend an existing commercial programming language and ones based on a mathematically cleaner language. The advantages of the former are familiarity to programmers and compatibility with legacy code. Examples of languages in this category are JML [9] and Spec# [2]. However, it can be extremely difficult to build a verification approach around such languages. Because they use reference semantics, they must include complex annotations to deal with subtle relationships such as object ownership and data abstraction [12,11]. As the need for user annotations appears to be an important factor inhibiting the adoption of these techniques, it is worth the effort to investigate alternative strategies. The use of mathematically cleaner languages can eliminate many of these complications, simplifying the semantics of the language and reducing the number of annotations required. Examples of languages in this category are SPARK [1] and Perfect [4].

Of the existing projects, only JML supports the full range of assertion-based verification approaches. There is currently no project aiming to support the full range of approaches for a mathematically clean language. Even in the case of JML, recent case studies [7] have applied the approaches separately, rather than in an integrated manner.

In this paper we present a new system called Omnibus [18,19] which supports runtime assertion checking, extended static checking and full formal verification for a mathematically clean language. The Omnibus language is superficially similar to Java but removes aspects that particularly complicate verification, in particular the use of reference semantics by default for objects. References are still needed to support polymorphism and so are used behind-the-scenes to implement the objects. However, they are not exposed to the programmer, with the language supporting a single equality operator that represents deep equality. These alterations simplify verification and eliminate many of the complications present in JML. For example, the specification of frame conditions [12] and the checking of invariants in the presence of callbacks [11] can be greatly simplified. A recent paper [8] highlights some particularly complicated examples that Java verification tools must be able to handle. These examples do not pose a problem for Omnibus, mainly because Omnibus outlaws the aspects of Java that are exploited to produce the difficult to verify examples.

The Omnibus language is supported by an IDE with a type checker, Java code generator, HTML documentation generator, and interactive and automated verifiers. Integrated support for different verification approaches is provided through a verification policy management system. Omnibus has been applied to a number of small and medium sized case studies.

A number of concepts from dependable systems development have natural definitions within Omnibus and Omnibus provides a range of facilities that can be used to support existing fault tolerant techniques.

Section 2 develops the case for using a cleaner language. Sections 3 and 4 present overviews of the Omnibus language and tools, respectively, Section 5 discusses dependable systems development with Omnibus and Section 6 concludes by discussing future work.

2. The case for using a cleaner language

There is much evidence of the difficulties of working with languages not specifically designed with verification in mind. An important cause of these difficulties is the use of reference semantics as the default mechanism for working with objects. Sophisticated techniques have had to be devised to control their use requiring extra tool support and additional programmer annotations. For example, there has been work on specifying frame conditions [12] and handling callbacks and invariants [11]. Leino notes in [10], "the reluctance to cope with the burden of annotating programs remains the major obstacle in the adoption of extended static checking technology into practice." As such, it would seem appropriate to attempt to remove any accidental difficulties such as complexities introduced by the language.

It is important to appreciate that the problems encountered in verifying modules written in these languages are not products of the formalization process itself, but are practical complexities that programmers have to grapple with when using the languages. The complexities of reference semantics frequently lead to a range of aliasing errors that can be difficult to detect.

Functional programming languages offer a cleaner basis for a verification approach than mainstream languages like Java because of their use of value semantics. However, functional languages have not been widely embraced by the software development industry.

An alternative approach is to take a mainstream language and adjust it to be more amenable to analysis. For example, SPARK removes aspects of Ada that are error-prone and complicated to verify. In SPARK, object variables hold values, not references hence naturally giving value semantics. However, in order to use inheritance, object variables must hold references so that dynamic binding can be supported. OO languages and value semantics are not incompatible but additional support is required to mask the use of references. Languages based on this approach can be relatively accessible to everyday programmers and not unnecessarily complicated to verify, though they might require programmers to adopt slightly different programming styles. Even JML, which uses reference semantics by default, provides the `pure` modifier that allows classes to be defined that obey value semantics.

The use of value semantics involves a trade-off of efficiency and expressiveness as well as reducing direct compatibility with legacy code. Our claim is that OO languages built on value semantics occupy an interesting position in the solution space involving an engineering trade-off worthy of investigation.

3. An overview of the Omnibus language

Omnibus is a new language that is similar to Java with adjustments making it more amenable to formal analysis. Like Java, it includes the concepts of packages, classes, methods, expressions, statements etc, but it also incorporates a behavioural interface specification language and uses value semantics for objects.

Similarly to Java, an Omnibus application consists of a set of *class definitions*. Each class contains a range of methods for manipulating instances of the class. There are three main types of method declaration in Omnibus: *constructors*, *functions* and *operations*. Constructors allow objects to be created, functions allow objects to be queried without side-effects and operations allow objects to be updated. The declaration of a method starts with a keyword identifying the type of method. Constructors are *class methods* whereas functions and operations are *object methods*.

In Omnibus, all objects are immutable with the system creating new objects behind-the-scenes as needed to preserve value semantics. This is hidden from the programmer who is allowed to think in terms of updating objects.

Specifications

Omnibus allows *heavyweight* specifications, which are suitable for full formal verification, as well as a *lightweight* specification style, suitable for run-time assertion checking or extended static checking.

Behaviour specifications: The behaviour of methods can be described using *behaviour specifications*. These are constructed from **`requires`**, **`changes`** and **`ensures`** clauses which give pre-conditions, frame conditions and post-conditions, respectively. A subset of the functions in the class is taken to represent the abstract state of the class. These are called *model functions*, are declared with the **`model`** modifier and do not have post-conditions. The behaviour of the other methods (the remaining functions along with the constructors and operations) is then defined in terms of them. When specifying operations, a **`changes`** clause is used to describe what model functions have their values changed and an **`ensures`** clause is used to describe how they are changed.

Requirements specifications: The requirements of a class are specified using **`initially`**, **`invariant`**, and **`constraint`** assertions. The **`initially`** assertions should hold over all freshly constructed objects, **`invariant`** assertions should hold over objects whenever they are accessible by code in other classes and **`constraint`** assertions should hold across any operation calls. Unlike the code-centric JML language, the requirements are not simply conjoined with the post-conditions of constructors and object methods and pre-conditions of object methods. Instead, they should follow from the behaviour specifications. This provides a useful way to verify the behaviour specifications, independent of an implementation.

```
spec class BankAccount {
     model function balance():integer
     model function overdraftLimit():integer
     function isOverdrawn():Boolean
            ensures result = (balance() < 0)
     function fundsAvailable():integer
            ensures result = overdraftLimit() + balance()
     constructor open(deposit:integer)
            requires deposit >= 0
            ensures balance() = deposit,
                         overdraftLimit() = 500
     operation deposit(amount:integer)
            requires amount >= 0
            changes balance
            ensures balance() = old balance() + amount
     operation withdraw(amount:integer)
            requires amount >= 0,
                         amount <= fundsAvailable(),
                         amount <= 300
            changes balance
            ensures balance() = old balance() – amount
     initially balance() >= 0
     invariant balance() >= -overdraftLimit()
     invariant overdraftLimit() >= 0
     constraint balance() >= old balance - 300
}
```

Fig 1. Heavyweight specification of a BankAccount class.

Example: Figure 1 presents a heavyweight specification for a simple Omnibus class modelling a `BankAccount`. A `BankAccount` is opened with an initial deposit and starts with an overdraft limit of 500. Given a `BankAccount`, you should be able to ask what the balance is, what the overdraft limit is, whether it is overdrawn and how much is available to be withdrawn. A `BankAccount` can be updated by depositing or withdrawing money. The requirements are that: (1) when an account is initially created, the balance should be at least zero, (2) the balance should never be more overdrawn than the overdraft limit permits, (3) the overdraft limit should never be negative, and (4) at most 300 can be withdrawn at one time. The `BankAccount` class is declared with the **`spec`** modifier which indicates that it defines only a specification and no implementation[1]. The **`old`** operator is used in the **`ensures`** clauses to refer to values from the pre-state.

Implementations

The public behaviour specification of a class should be defined in terms of a set of model functions, without making reference to implementation details. In contrast, the implementation of the class is solely defined in terms of private *attributes*. Each of the model functions must then be implemented at the private level in terms of the attributes. Method implementations are defined using a Java-style implementation language containing an assignment statement, operation call statements, a declaration statement, an assert statement, an if statement, a for loop and a while loop. Loops can be annotated with loop invariant assertions.

Inheritance

Omnibus supports single behavioural inheritance. A class inherits all the requirements of its superclass and can choose to either implicitly inherit or explicitly override the methods in the superclass. The overriding method definitions can give different behaviour specifications with weakening of the pre-condition (i.e. the **`requires`** clause) and strengthening of the post-condition (calculated from the **`changes`** and **`ensures`** clauses) permitted. Model functions can also be redefined as derived functions but when this is done, methods inherited from the superclass must have their behaviour redefined in terms of the new model functions.

[1] JML uses the **`model`** modifier to signify this whereas we use that for a different concept.

Libraries

Like JML, Omnibus hides mathematical abstractions like sequences and sets behind a façade of library classes. Users interact with these classes through methods just like any other class, and do not need to learn additional mathematical notation to manipulate them. This is in contrast with the Perfect language which provides support for sequences, sets etc in the language itself. I/O is also achieved through the libraries. This centres around a uniquely typed [16] `Environment` class which is passed into the application at the entry point.

Limitations

There are limitations in the current version of the language. Some of these limitations, such as the removal of static data, have been purposefully introduced to simplify verification. Support for exceptions and Java-style interfaces is under development. We do not currently handle arithmetic overflow, concurrency or termination.

4. The Omnibus IDE

The Omnibus IDE incorporates standard facilities for managing files and projects and uses a jEdit component to support syntax highlighting and bracket matching while editing source code. In addition to this, it provides a type checker, a Java code generator incorporating run-time assertion check generation, an HTML documentation generator, a static verifier supporting extended static checking and full formal verification, and, most importantly, a Verification Policy Manager [19], which provides a flexible means for integrating the use of the different verification approaches within different parts of a system.

Figure 2 presents an overview of the Omnibus static verifier. It takes as input an Omnibus project consisting of a collection of source files, referenced jar files and a *verification policy* describing what level of verification should be performed on each file. The files are then parsed and type checked before being passed to the static verifier. The verifier uses two theorem provers: the interactive PVS prover [15] and the fully automated Simplify prover [5]. The first step in the process is to translate the classes in the source files and referenced jar files into the logics of the two theorem provers. The static verifier then uses two generic modules: a specification verifier and a symbolic executor, to generate VCs over the translated specifications. The specification verifier generates VCs to check that the behaviour of heavyweight specifications satisfies their requirement specifications. The symbolic executor executes implementations using symbolic values to check that implementations satisfy their behaviour specification. The VCs are expressed in an extension of the Omnibus assertion language and can then be translated into either PVS or Simplify conjectures depending on the verification strategy specified in its verification policy. Finally, the generated files are passed to the corresponding provers. In the case of PVS, the user must manually launch the prover and attempt to verify the conjectures. In contrast, the tool is able to automatically invoke the Simplify prover and process its responses to give user friendly error messages.

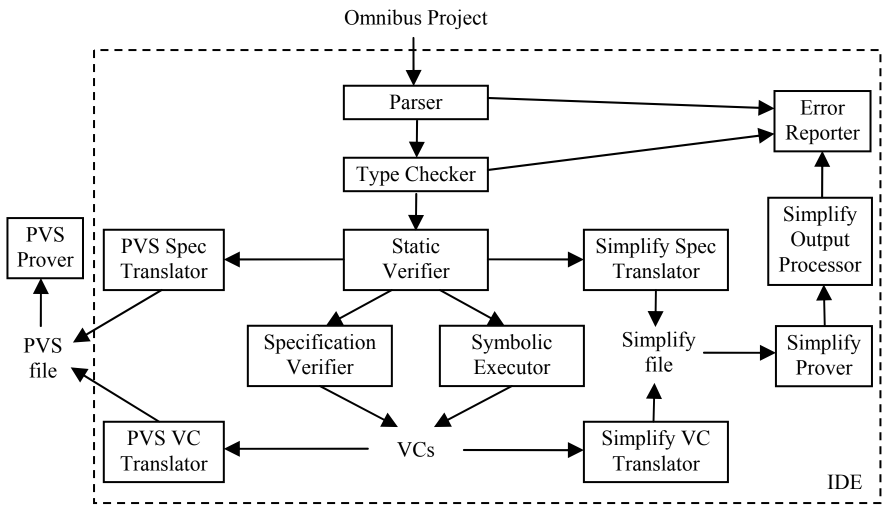


Fig. 2. Diagram of the Omnibus IDE's static verification process.

We were guided through many of the practical obstacles by the writings of Jacobs et al. [7], Cok [3] and Leino [13]. We were able to make a number of simplifications over their approaches due to our use of the simpler Omnibus language. For example, we do not require any formal modeling of the heap.

5. Dependable systems development

A system is said to have a *failure* if the service it delivers deviates from the desired behaviour. Such failures are caused by flaws in a system called *faults*. These faults can be present in the hardware, software or non computer-based parts of the system.

It can be difficult to precisely define what is meant by failure, fault and 'desired behaviour'. There are natural definitions for each of these within the Omnibus framework. In Omnibus, dynamic assertion violation errors indicate faults and static assertion violation warnings indicate possible faults. If the assertion violations are concerned with the top-level specification of the system then they are failures. This fits nicely alongside the idea of Heimerdinger [6] of viewing faults as failures in other systems which interact with the system under consideration. The accepted approach for defining 'desired behaviour' is to use a specification. The Omnibus language allows specifications to be defined precisely.

The development strategies currently at our disposal are unable to consistently produce realistic systems without faults. The root cause of these problems is complexity: of the systems being developed and of the techniques used to develop them. By using a semantically simpler language, Omnibus aims to reduce the amount of unnecessary complexity, allowing the essential difficulties to be focused on. While it is unavoidable that realistic systems will contain some faults, we still want these systems to be *dependable* i.e. be trustworthy enough that reliance can be placed on the service they deliver.

There are different means of attaining dependability of systems; among them are *fault avoidance* and *fault tolerance*. Fault avoidance is concerned with preventing the introduction of faults as the system is being developed. This is achieved through the use of quality control techniques during the specification, design and implementation of a system. Omnibus provides a rigorous framework for performing such quality control of software development, allowing the consistency of specifications, designs and implementations to be formally verified. A key strength is that the separation of behaviour and requirement specifications allows the internal consistency of specifications to be verified independent of any implementation. Fault removal is a related approach where verification and testing techniques are used to locate faults in a system once it is developed. The traditional choices are testing or full formal verification. However, testing techniques do not cope well with the large state spaces of realistic systems and heavyweight verification is typically too costly to use for systems in their entirety, particularly those implemented using semantically complex languages such as Java and C#. Omnibus helps address this by supporting a range of verification techniques from the dynamic and static checking of lightweight assertions to full formal verification relative to heavyweight specifications. Omnibus can also be used to express test harnesses in terms of symbolic input values and these can be verified using symbolic execution. Such test scenarios allow for greater error coverage and can be equivalent to large numbers of concrete test scenarios.

Fault tolerance is concerned with maintaining the correctness of the delivered service in the presence of faults. Fault tolerance can be applied at three different levels: *hardware fault tolerance*, *software fault tolerance* and *system fault tolerance* [6,17].

Hardware fault tolerance is concerned with compensating for faults in the low-level computing hardware of a system and is beyond the scope of Omnibus.

Software fault tolerance involves the structuring of a computer system to compensate for faults in the software system itself. Omnibus provides a range of facilities that can be used to support existing fault tolerant techniques. Its key strengths are its expressive specification language and support for the automatic generation of run-time checks from assertion annotations. There are two groups of software fault tolerant techniques, those that aim to tolerate faults in single software modules (single-version techniques) and those that employ redundant software modules (multi-version techniques).

Single-version techniques include detection and containment techniques. Omnibus assertion checks greatly aid fault detection. Fault detection is traditionally carried out through acceptance tests such as *reasonableness checks* and *structural checks*. Omnibus assertions are a natural way of expressing some of these acceptance tests. For example, reasonableness checks map nicely to behaviour specifications and structural checks equate to invariants. Omnibus run-time assertion checks also allow faults in an executing program to be detected earlier than they would otherwise be. This is because in Omnibus faults are detected as soon as one of the assertion checks fails rather than at some later point when a run-time error is triggered or an acceptance check is failed. The advantage of detecting faults earlier is that it reduces the amount of damage that they can do. Fault containment techniques are also supported by Omnibus. The Object-Oriented facilities of the language provide support for modularization and specifications allow the situations where actions are permissible to be explicitly defined, preventing a faulty component from making an invalid invocation of another component.

Multi-version techniques employ redundant modules to provide fault tolerance. Hardware fault tolerance is relatively well understood and utilises redundancy heavily to cope with low-level production errors. However, these techniques do not map directly to the software domain. Simply duplicating a software component does not help address software faults since all copies of the component will have identical faults. To get around this, different but equivalent implementations of a component can be created. These implementations must be developed independently so that they do not share common faults. This process is called *design diversity*. Just as the entire system will not typically merit the use of full formal verification, it will not typically be justifiable to design multiple versions of the complete system. As with the verification policy management strategy, the class (or perhaps even the method) is a more appropriate scale to operate on. It is also important that the diverse designs are equivalent. Omnibus can be used to demonstrate that the designs satisfy a common specification, detecting inconsistencies early on.

Finally, system fault tolerance involves the development of facilities to compensate for failures in parts of the system that are not directly computer-based e.g. external devices such as sensors. By using suitable specifications for these external devices, Omnibus can be used to statically verify that a computer system copes with every eventuality e.g. sensors operating correctly and sensors failing. This form of static verification of a-priori known potential faults is of course possible using other formal frameworks.

6. Future work

Work on the language is currently focusing on the handling of equality of objects. There are a number of features that we wish to add to the Omnibus language. These include support for predicate subtypes, enumeration types and exceptions. The libraries are the area needing most work. In particular, we would like to add support for GUIs, File I/O and XML. The IDE is largely finished, needing only a number of refinements. Two key problems we have met are limits of the expressiveness of our assertion language and the level of repetition between a heavyweight specification and a corresponding implementation. We are currently working to address these problems and to develop larger case studies.

7. References

1. J. Barnes – “High Integrity Software: The SPARK Approach to Safety and Security”, Addison-Wesley, 2003.
2. M. Barnett, K.R.M. Leino, W. Schulte – “The Spec# programming system: An overview”, in the proceedings of CASSIS 2004, Springer LNCS 3362, 2005.
3. D.R. Cok – “Reasoning with specifications containing method calls in JML and first- order provers”, Formal Techniques for Java-like Programs workshop at ECOOP, 2004.
4. D. Crocker – “Safe Object-Oriented Software: the Verified Design-by-Contract paradigm”, Procs. of the 12th Safety-Critical Systems Symposium, Springer-Verlag, 2004.
5. D. Detlefs, G. Nelson, J.B. Saxe – “Simplify: A theorem prover for program checking”, Technical Report HPL-2003-148, HP Labs, 2003.
6. W. Heimerdinger, C. Weinstock – “A Conceptual Framework for System fault Tolerance”, Technical Report CMU/SEI-92-TR33. ESC-TR-92-033. SEI. October 1992.
7. B. Jacobs *et al.* – “Formal verification of a commercial smart card applet with multiple tools”, Proceedings of AMAST 2004, Springer LNCS 3116, 2004.
8. B. Jacobs *et al.* – “Java Program Verification Challenges”, Proceedings of Formal Methods for Components and Objects, Springer LNCS 2852, 2003.
9. G.T. Leavens *et al.* – “Preliminary Design of JML: A Behavioral Interface Specification Language for Java”, Dept. of Computer Science, Iowa State University, TR #98-06p, 2003.
10. K.R.M. Leino – “Extended Static Checking: A Ten-Year Perspective”, Informatics—10 Years Back, 10 Years Ahead, Springer LNCS 2000, 2001.
11. K.R.M. Leino, P. Muller – “Object invariants in dynamic contexts”, ECOOP 2004 — Object-Oriented Programming, Springer LNCS 3086, 2004.
12. K.R.M. Leino, G. Nelson – “Data abstraction and information hiding”, ACM Trans-actions on Programming Languages and Systems, 24(5):491–553, September 2002.
13. K.R.M. Leino, J.B. Saxe, C. Flanagan – “The logic of ESC/Java”, http://research.compaq.com/SRC/esc/design-notes/escj08a.html
14. B. Meyer – “Eiffel : The Language”, ISBN 0132479257, Prentice Hall, 2000.
15. S. Owre *et al.* – “PVS: Combining Specification, Proof Checking, and Model Checking”, Proceedings of CAV 1996, Springer LNCS 1102, 1996.
16. M.J. Plasmeijer – “CLEAN: a programming environment based on Term Graph Rewrit-ing”, Proceedings of SEGRAGRA'95, ENTCS 2, 1995.
17. W.Torres-Pomales – “Software Fault-Tolerance: A Tutorial”, NASA/TM-2000-210616, October 2000.
18. T. Wilson – Omnibus home page. Available at http://www.cs.stir.ac.uk/omnibus/
19. T. Wilson, S. Maharaj, R.G. Clark – “Omnibus Verification Policies: A flexible, configur-able approach to assertion-based software verification”, accepted for publication in SEFM 2005, Koblenz, Germany, September 2005.

Towards Formal Development of Mobile Location-Based Systems

Alexei Iliasov[1], Linas Laibinis[2], Alexander Romanovsky[1], Elena Troubitsyna[2]

[1]Newcastle University, UK. {Alexei.Iliasov, Alexander.Romanovsky}@ncl.ac.uk
[2]Åbo Akademi, Finland. {Linas.Laibinis, Elena.Troubitsyna}@abo.fi

1. Introduction

1.1 Motivation

Mobile agents have many attractive features to offer and they are often mentioned as a future mainstream industry-level software technology. The agent technology naturally solves the problem of decoupling complex software into smaller parts that are easier to design, code and maintain. It helps to use distributed computing power effectively while hiding many of the details and complexities of a hosting environment. Recent advances in mobile computing and wireless networks lead to introduction of host (physical) mobility that offers totally new opportunities and as well raises new problems. Though substantial research has been conducted on developing middleware solutions supporting mobile agents, the mobile agent technology is still not mature enough to become a practice in industrial software development. There are several areas in which no general solutions have been found yet. One of them is ensuring interoperability of independently designed agents and correctness of the overall mobile system. In this work we will present a background for building a formal development methodology that addresses this problem.

Agent software is designed to interact with other agents during its lifetime. Most research in the area discusses only centralized development process, when all the participating pieces of software (code of the agents) are created at the same site to solve common problems. In this case agents are mostly useful as a replacement of conventional client-server scheme with migrating clients or/and servers. However the application area of the mobile agents is much broader and, to make full use of their communication and migration capabilities, we need to assume systems are composed dynamically out of agents developed independently at different sites and for different purposes. Such configurations are impossible if agents are merely anonymous black boxes. In our view, to cooperate, agents must be based upon some common specification of their functionality. This specification should be formally developed and verified to ensure the desired properties of the application composed of agents. Developers of individual agents can independently extend the specification (using a refinement method) to add unique features without losing compatibility with other agents derived from the same specification.

The specification should be minimal in a sense that it does not have to provide many design details but it should be complete enough to identify what services the agent has to offer and what services it is looking for. This information should describe how to communicate with the particular class of agents, what such agents expect as input, and what output they produce.

1.2 Background

Mobile agent systems are often symmetric in a sense that each system participant roughly carries the same middleware implementation. Agents can dynamically and autonomously form new groups and communicate. However in this paper we explore an asymmetric approach in which different parts of the system carry different basic functionality. One particular example of such view is *a location-based* scheme. In this model locations provide services to the agents, such as connectivity and coordination space. Agents are not able to communicate with each other without a location support. The choice of the scheme is supported by the fact that the majority of the mobile applications assume that agents meet in physical or logical locations providing a set of designated services to them. Hence, the asymmetric scheme is closer to the traditional service provision architectures. It can support large-scale mobile agent networks in a very predictable and reliable manner. It makes better use of the available resources since most of the operations are executed locally. Moreover, location-based architecture eliminates the need for employing complex distributed algorithms or any kind of remote access. This allows us to guarantee atomicity of certain operations without sacrificing performance and usability. This scheme also provides a natural way of introducing context-aware computing by defining location as a context. The main disadvantage of the location-based scheme is that an additional infrastructure is always required to support mobile agent collaboration.

The *coordination paradigm* (originated in Linda [4]) has become the dominating environment in which a number of mobile systems are built (including Lime [7], Klaim [2], etc.). Linda is a set of language-independent coordination primitives that can be used for communication and coordination between several independent pieces of software. First used for parallel programming, it later became a core component of many mobile software systems because it fits nicely the main characteristics of the mobile systems: openness, dynamicity, anonymity of agents and their loose coordination. Linda-based coordination systems specifically designed for mobile applications supporting both physical mobility, such as a device with running application travelling along with its user across network boundaries, and logical mobility, when a software application changes its hosting environment.

The rest of the paper is organized as follows. Section 2 introduces a number of basic abstractions to be used in development of mobile systems. Sections 3 describes a rigorous development process supporting these abstractions. Section 4 presents a formal abstract specification of the middleware. Finally, the last section presents conclusions and outlines our future work.

2. System structure

The CAMA (**context-aware mobile agents)** system consists of a set of locations. Active entities of the system are agents. An agent is a piece of software that meets a number of requirements. Each agent is executed on its own platform. The platform provides execution environment interface to the location middleware. Agents communicate only with other agents in the same location. Agents can migrate from location to location logically (connections and disconnection) or physically (e.g. movement of a PDA on which the agent is hosted on). They can also logically migrate from platform to platform using weak code mobility. Compatible agents collaborate through a scoping mechanism. A scope defines a joint activity of several agents. The scoping mechanism also isolates non-compatible agents from each other. Below are the details of the introduced concepts.

A ***location*** is a container for scopes. It can be associated with a particular physical location and can have certain restrictions on the types of supported scopes. It is the core part of the system as it is provides means of communications and coordination between agents. Location is a named entity and for simplicity we assume that each location has a unique name in the given context. This roughly corresponds to IP addresses of hosts on network which are often unique in some local sense. Location must keep track of present agents and their properties in order to be able to automatically create new scopes and restrict access to existing ones. The more detailed location description is presented in the form of a formal specification (see Section 4).

Certain locations may prevent agents from entering without an authorization. To be allowed to enter a location, an agent must have a key issued by it. Keys may be permanent or have a validity period determined by the issuing location Agent must have to acquire a key on a different location before entering a protected location.

Locations may provide special services, like access to a service from a variety of devices connected to the location, making enquires and so on. Each Location may have its own unique set of services and provided operations. They are made available to agents via what appears to agents as a normal scope though some roles in these scopes are implemented by the location system software. As with all scopes, agents are required to implement specific interfaces in order to connect to a location-provided scope. An example of such services includes printing on a local printer, access to Internet, making a backup to a location storage, migration and etc. In addition to supporting scopes as mean of agent communication, location may also support logical mobility of agents, hosting of platforms and agent backup. Hosting of platform on a location allows agent to execute without a PDA. For example, a user may decide to move an agent from his PDA to a location before leaving the location with his PDA. In addition to the above, location, by a request from an agent, may play in certain types of scopes a role of a trusted third party that is neutral to all the participating agents. This facilitates implementation of various transaction schemes.

A ***platform*** provides an execution environment for an agent. It is composed of a virtual machine for code execution, networking support and middleware for interaction with location. A platform may be supported by PDA, smart-phone, laptop or a location server. The concept of platform is important to clearly differentiate between a location providing coordination services to agents and middleware that only supports agent execution. In other approaches no such distinction is made.

An ***agent*** is a piece of software implementing a set of roles which allow it to take part in certain scopes. All agents must implement the minimal functionality called the default role, which specifies activities outside scopes.

A ***scope*** is a dynamic container for tuples. It provides an isolated coordination space for compatible agents by restricting visibility of the tuples contained in the scope to the participants of the scope. Scopes are initiated by an agent and then atomically created by Location when all the participants are ready. Scopes can be nested and scope participants can create new contained scopes. Scope is defined by the set of roles and a set of logical restrictions.

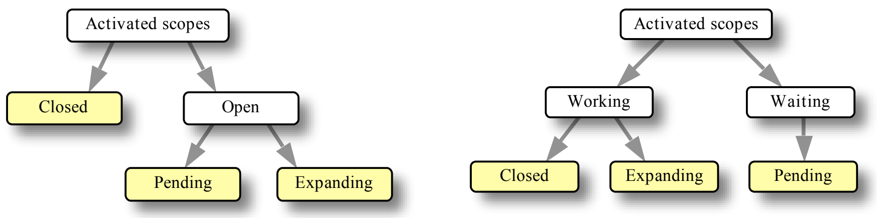


Fig. 1. Scope classification a) according to the availability of scopes for new agents and b) according to the agent activity in a scope.

A scope becomes *activated* after some agent creates it with the *CreateScope* operation. A scope is *open* when there are some vacant roles in it, and is *closed* when all the roles in it are taken. A scope is *pending* if some required roles are not taken yet and *expanding* if all the required roles are taken but there still some vacant roles. Closed and expanding states correspond to working scopes, where agents can communicate. All participants of a pending scope are blocked until the scope state is changed into closed or expanding.

A ***role*** is an abstract description of agent functionality. Each role is associated with some scope type. An agent may implement a number of roles and can also play several roles in the same scope or different scopes. There is formal relationship between a scope and a role of a scope.

Introduction of scopes and roles offers agents an entirely new way to discover each other and to collaborate with each other. After arrival to a new location, an agent looking for partners, initiates scope creation or join protocol. They are implemented as a request to the controlling system (middleware) to find appropriate partners ready for certain type of activity. In a request agent specifies type of scope it wants to work in and a role it is going to take. The system then creates a scope or finds an existing matching scope with available role for the agent. This procedure is executed atomically. As soon as all the required roles are taken, the system creates a separate coordination space for the group of agents participating in the scope. Isolation achieved this way greatly simplifies agent design since while in a scope agent may safely assume reasonable behaviour of their partners. In a scope agents remain anonymous as long as they need and procedures of scope joining or creation do not change this.

The CAMA approach supports the context-awareness of mobile agents. The context of an agent in CAMA systems consists of is composed of the following parts: a set of locations the agent is connected to, the state of scopes in which the agent is currently participating (including tuples contained in these scopes) and role attributes of other agents in collaborating with the agent.

3. Formal Development Process

Formal development process of the CAMA system consists of several steps. First, we create abstract specifications of the middleware (location) and the scopes that will be supported by the system. Then we develop (by the stepwise refinement method) specifications of different roles participating in scopes. Finally, we compose an agent specification as a combination of several developed roles (i.e., agent interfaces) and the default functionality defining the agent behaviour outside scopes.

The agent specification can be further refined adding more details and custom functionality. Compatibility of different agents is ensured by the fact that all agents have been developed by the formal refinement method from the same abstract specifications of different roles and the middleware. Therefore, agents can collaborate making safe assumptions about the functionality of their peers.

In the next subsection we give a brief introduction into our formal framework – the B Method, which we will use to formalise the development process described above.

3.1 The B Method

The B Method [1] (further referred to as B) is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [6]. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [8], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

The development methodology adopted by B is based on stepwise refinement [3]. While developing a system by refinement, we start from an abstract formal specification and transform it into an implementable program by a number of correctness preserving steps, called *refinements*. A formal specification is a mathematical model of the required behaviour of a (part of) system. In B a specification is represented by a set of modules, called Abstract Machines. An abstract machine encapsulates state and operations of the specification and as a concept is similar to a module or a package.

Each machine is uniquely identified by its name. The state variables of the machine are declared in the VARIABLES clause and initialized in the INITIALISATION clause. The variables in B are strongly typed by constraining predicates of the INVARIANT clause. All types in B are represented by non-empty sets. We can also define local types as *deferred sets*. In this case we just introduce a new name for a type, postponing actual definition until some later development stage.

The operations of the machine are defined in the OPERATIONS clause. In this paper we use Event B extension of the B Method. The operations in Event B are described as guarded statements of the form $\boxed{\text{SELECT cond THEN body END}}$. Here cond is a state predicate, and body is a B statement. If cond is satisfied, the behaviour of the guarded operations corresponds to the execution of their bodies. However, if cond is false, then the execution of the corresponding operation is suspended, i.e., the operation is in waiting mode until cond becomes true.

The generalised version of the guarded operation is ANY operation. The syntax of ANY operation is $\boxed{\text{ANY vars WHERE cond THEN body END}}$. The operation corresponds to a family of events or a parameterised event operation. It is triggered by any acceptable values of the variables vars satisfying the condition cond. The variables vars are then used as local variables in the operation body.

B statements that we are using to describe a state change in operations have the following syntax:

```
S  ==  x := e  |  IF cond THEN S1 ELSE S2 END | S1 ; S2  |  x :: T  |
                 S1 || S2 | ANY z WHERE cond THEN S END  |  ...
```

The first three constructs – assignment, conditional statement and sequential composition (used only in refinements) have the standard meaning. The remaining constructs allow us to model nondeterministic or parallel behaviour in a specification. Usually they are not implementable so they have to be refined (replaced) with executable constructs at some point of program development. The detailed description of the B statements can be found elsewhere [1].

3.2 Development of Scopes and Roles

The specification of a scope describes general functionality of several collaborating agents (in particular roles). The task of formal development is to use the specification as the starting point for the derivation of specifications of the corresponding agent roles (interfaces). To guarantee correctness of the resulting role specifications, we use formal refinement and decomposition techniques. For example, Fig.2 shows that the Lecture scope is decomposed into roles Student and Teacher defining functionality of the corresponding agents.

On the other hand, we have to take into account scope nesting, when scopes have embedded subscopes providing some extended functionality. Subscope specifications can be naturally derived from the original scope specification via refinement. After verifying the correctness of refinement, we can continue the development process by decomposing the specification into corresponding roles as described above. In Fig.2, we show how scope Lecture is refined by subscope Group work, which is consequently decomposed into roles Student' and Teacher'.

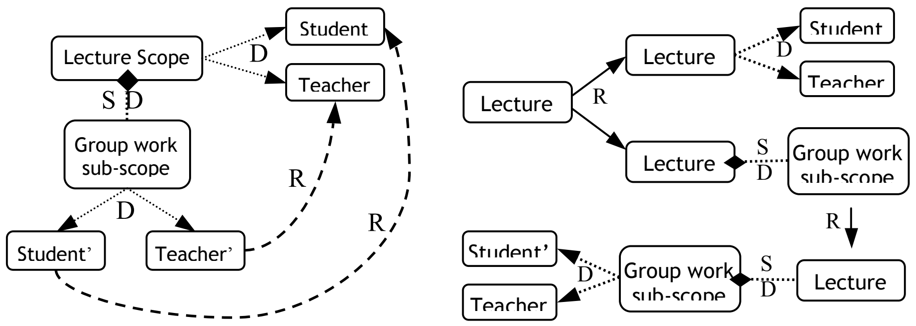


Fig. 2. a) Orthogonal decomposition diagram b) its representation as a parallel refinement. SD is scope decomposition; D – decomposition of a scope into roles; R – refinement.

As a result, we have two orthogonal development processes with the same starting point – the original specification of a scope. Both developments arrive at role specifications describing agent functionality in the corresponding scopes. However, the hierarchy of scopes and subscopes should be reflected in the corresponding specifications of agent roles. Hence the roles in subscopes must be the extensions of the corresponding roles in the scopes. In other words, to guarantee the consistency of developed roles, we have to show that the subscope roles refine the corresponding scope roles.

In our Lecture scenario, we derived the specifications of agents in roles Student and Teacher. These specifications describe the functionality of the corresponding agents after joining scope Lecture. On the other hand, roles Student' and Teacher' describe the behaviour of the corresponding agents while they enter scope Group Work which is a subscope of Lecture. These roles have to satisfy the requirements specified in Student and Teacher. At the same time, they can provide additional functionality specific to Group Work. By proving formally that Student' is a refinement of Student, and Teacher' is a refinement of Teacher, we guarantee consistency of agent behaviour in nested scopes Lecture and Group work. In Fig.2, this is shown by the arrows connecting roles Student' and Student, and roles Teacher' and Teacher.

3.3 Agent Design

Agent design starts with the selection of roles that the agent must implement. It is permitted to implement any number of roles from different scopes. Initially roles inside of an agent are totally independent specifications that may well correspond to several independent processes running in an agent. Agent refinement specifies additional operations that control agent behaviour during migration, location selection, scope creation and joining, and other activities not covered by roles.

During agent refinement process, the agent roles can also be refined, possibly by adding some new functionality. Due to the nature of refinement, the refined roles are still compatible with the original abstract roles.

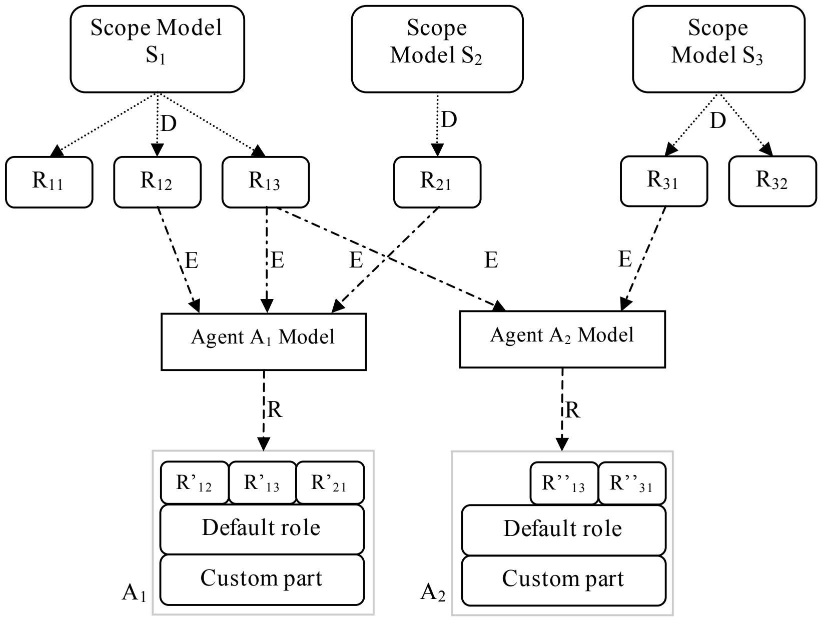


Fig.3. Relations between agents, scope models and roles. D – decomposition of a scope into roles; E – extension of role specification an agent model; R – refinement of an agent model.

We start building an agent specification by extending one or more roles obtained formally through the decomposition of abstract scope models (see Fig. 3). The refinement step introduces a specification of the minimal agent functionality called the default role. It allows an agent to talk to locations, create/join/leave scopes, and migrate. The agent may also need some logic that glues independent interfaces and allows them to talk to each other. This is done via the global agent variables and the special methods for accessing to them.

After the agent specification is ready, it is used to build the source code for the actual agent program. The source is linked against the middleware library to get an executable agent program. The generated agent source may run on PDAs, laptops, desktop PCs and smart-phones using the platform-specific middleware implementation as the adaptation layer.

The standard work cycle of an agent looks like this: an agent detects the available locations and connects to at least one of them, then looks for current activities on the location(s) or creates its own new scope, and finally joins a scope and plays one of the implemented roles in it. Only when the agent decides to play a particular role in a scope, it really starts to cooperate with other agents. The agent is capable of understanding its peers since the role functionalities of all the scope participants are based on the same abstract model. As a result, the composition of agent functionalities in a scope corresponds to the initial abstract model.

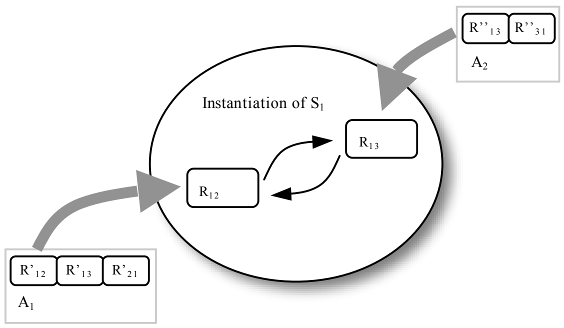


Fig. 4. An instantiation of an abstract model

The correctness of a model instantiation, or in other words, the fact that the scope instantiates the corresponding abstract scope model, can be demonstrated by analysing the agent design process and assuming that there is a correct transition from agent model to agent implementation. In Fig.4 we illustrate an instantiation of an abstract model which is formed when all the roles in the scope are taken by some agents.

3.4 Fault Tolerance

Ability to operate in a volatile, error prone environment will be an intrinsic feature of CAMA. Hence CAMA systems should be able to withstand various kinds of faults, i.e., guarantee fault tolerance. The most typical fault is a temporal connectivity loss which can cause failures of communication between cooperating agents or between an agent and the location.

Since in the CAMA approach the agent and location software are developed from the corresponding B specifications, the fault tolerance mechanisms should be already integrated into these specifications, so that development of fault tolerance means is becoming part of the system development. For example, while modelling collaboration between agents in the specification of a scope, we have to define the agent behaviour in the presence of message losses, hardware failures etc. Moreover, while developing agent roles (interfaces) from the corresponding scope specifications, fault tolerance mechanisms should be distributed between involved parties.

Representing fault tolerance in CAMA constitutes an important research topic which we will further investigate in our future work.

4 B Specification of the Middleware

To ensure correct behaviour of the location-based system, the middleware of the location should enforce a certain discipline on agents. For instance, the properties of the scopes defined upon scope creation are preserved in spite of volatile connectivity and dynamic nature of scopes. Moreover, it should guarantee the integrity of the information about agents in locations and scopes. These complex interdependencies should be stated explicitly and verified. We have developed a formal specification of the location middleware which is the core of the system. It corresponds to the most complex part of the system and not only defines the operations that the location provides to support communication between agents but also state the properties of the data structures in the location. The actual middleware implementation will be based upon this formal model. An abstract description of the location specification is presented below. The full B specification can be found in [9].

```
MACHINE
   Location
VARIABLES
   AgentNames,         /* Agents active in the location */
   Scopes,             /* Created scopes */
   ScopeRolesTaken,   /* A number of agents taken a particular role in a particular scope */
   AgentRoleData,      /* Public data disclosed by the agent while taking a certain role */
   AgentScopes,        /* For each active agent defines the scopes in which it is active */
   ScopeAttributes,    /* Scope descriptions provided by scope creators */
   ScopeAgentRoles     /* The roles taken by agents in active scopes */
INVARIANT
   Types of variables & interdependencies between data

INITIALIZATION
  Initially there are no agents and correspondingly no scopes in the location
  ...
OPERATIONS

/* Engagement request */
a_id < --Engage =
  ANY Role_and_Data WHERE
     Role_and_Data is the information about the supported roles supplied by the agent
  THEN
    CHOICE
      successful engagement to the location by issuing valid ID to the agent via a_id and
      update of AgentNames and AgentRoles
    OR
      failed engagement to the location by issuing invalid ID to the agent
    END;
  END;

/* Disengagement request */
```

```
rr <-- Disengage = …

/* Scope creation request from an agent */
scope_id <-- CreateScope =
   ANY a_id, scopeDescr, role WHERE
       a_id is ID of the agent requesting to create a scope
       scopeDescr defines the necessary conditions for joining a scope
       role: the role that the requesting agent a_id will play in the created scope
   THEN
      CHOICE
          successful scope creation by issuing valid scope ID via scope_id,
          updating list of active scopes Scopes and list of
         scope descriptions ScopeAttributes updating AgentScopes,
          ScopeRolesTaken and ScopeAgentRoles
      OR
         unsuccessful scope creation by issuing invalid scope ID via scope_id
      END
  END;

/* Scope remove request */
result <-- DeleteScope = …

/* Scope join request */
result <-- JoinScope =
  ANY a_id, scope_id, role WHERE
      a_id is ID of the agent requesting to join the scope
      scope_id is ID of the scope which the agent is attempting to join
      role is the role which a_id will play in the scope
  THEN
     IF
        the agent a_id is not already participating in scope_id &
        requested role is a valid role for the scope &
        conditions for participating in the scope are not violated
     THEN
        the agent a_id is successfully joined the scope
        the information about the agent is updated
        in AgentScopes, AgentRoles, and ScopeAgentRoles
        the information about the number of agents playing the role is updated for the scope
     ELSE
        the agent a_id is rejected to join the scope
     END
  END;

/* Scope leave request */
result <-- LeaveScope = …

/* Prompt information about the scopes in which an agent can participate */
scopes <-- GetScopes = …
END
```

5 Conclusions

The presented work is tightly linked to the Ambient Campus case study of the RODIN Project. One of the project goals is to develop the methodology (based on formal methods) that would allow us to fully model and build the mobile location-based systems. The requirements document (written for the Ambient Campus case study) is the first step towards creating the formal model of such systems.

At the same time, we are developing middleware that will support our mobile agent abstractions. This paper presents the formal B specification of the location, i.e., the core part of the middleware. The choice of the location-based architecture (discussed in [5]) has influenced all the parts of our work on the case study, including the methodology.

It is our plan to investigate more closely the agent design process. We are also planning to conduct several extensive experiments covering the full cycle of system development – starting from an abstract system model through all steps until we get running software.

Acknowledgments. This work is supported by IST FP6 RODIN Project.

References

1. J.-R. Abrial. *The B-Book*. Cambridge Univ. Press, 1996.

2. R. De Nicola, G. Ferrari, R. Pugliese. *Klaim: a Kernel Language for Agents Interaction and Mobility*. IEEE Transactions on Software Engineering, 24(5):315-330, 1998.

3. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

4. D. Gelernter. *Generative Communication in Linda*. ACM Computing Surveys. 7(1): 80-112, 1985.

5. A. Iliasov, A. Romanovsky. *Exception Handling in Coordination-based Mobile Environments*. Proc. of COMPSAC 2005. Edinburgh, (UK), July 2005. IEEE CS.

6. *MATISSE Handbook for Correct Systems Construction*. 2003.http://www.esil.univ-mrs.fr/~spc/matisse/Handbook/

7. G. P. Picco, A. L. Murphy, G.-C. Roman. *Lime: Linda Meets Mobility*. Proc of the 21st Int. Conference on Software Engineering (ICSE'99), Los Angeles (USA), May 1999.

8. Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 2001. Available at http://www.atelierb.societe.com/index.html

9. B Specification of Location. Available from http://www.abo.fi/~Linas.Laibinis/Location.mch

Examples of how to Determine the Specifications of Control Systems

Joey W Coleman and Cliff B Jones

School of Computing Science
University of Newcastle upon Tyne
NE1 7RU, UK
email: {j.w.coleman,cliff.jones}@ncl.ac.uk

Abstract. Creating the specification of a system by focusing primarily on the detailed properties of the digital controller can lead to complex descriptions that have no coherence. An argument put forward in a recent paper by Hayes, Jackson, and Jones gives reasons to focus first on the wider environment in which the system will reside. This paper informally explores two examples so as to illustrate this approach to system specification.

1 Overview of approach

The general idea of the "Hayes/Jackson/Jones" approach [HJJ03] is simple: for many technical systems it is easier to derive their specification from one of a wider system in which physical phenomena are measurable. Even though the computer cannot affect the physical world directly, it is still worthwhile to start with the wider system. The message can be stated negatively: don't jump into specifying the digital system in isolation. If one starts by recording the requirements of the wider (physical) system, the specification of the technical components can then be *derived* from that of the overall system; assumptions about the physical components are recorded as rely-conditions for the technical components.

In order to be able to write the necessary specifications, some technical work derived from earlier publications of Hayes, Jackson and Jones has to be brought together. The process of deriving the specification of the software system involves recording assumptions about the non-software components. These assumptions are recorded as rely conditions because we know how to reason about them from earlier work on concurrency (e.g. [Jon81,Jon83,Jon96]). In most cases, we need to reason about the continuous behaviour of physical variables like altitude: earlier work by Hayes (and his PhD student Mahony) provides suitable notation [MH91]. The emphasis on "problem frames" comes from Jackson's publications [Jac00].

A trivial example of the HJJ approach is a computer-controlled temperature system: one should not start by specifying the digital controller; an initial specification in terms of the actual temperature should be written; in order to derive the specification of the control system, one needs to record assumptions (rely-conditions) about the accuracy of sensors; there will also be assumptions about

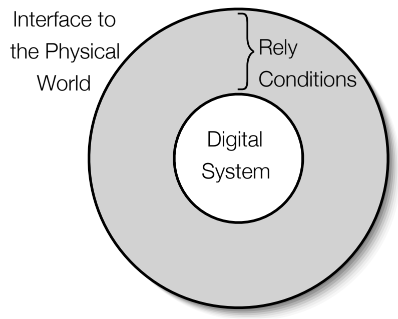


Fig. 1. Bridging from the physical world to a digital control system

the fact that setting digital switches results in a change in temperature. Once the specification of the control system has been determined, its design and code can be created as a separate exercise. At all stages — but particularly before deployment — someone has to make the decision that the rely conditions are in accordance with the available equipment. Figure 1 gives an abstract view of the HJJ approach. The referenced [HJJ03] outlines this procedure for a "sluice gate" controller. The analysis includes looking at tolerating faults by describing weaker guarantees in the presence of weaker rely conditions.

Notice that it is not necessary to build a complete model of the physical components like motors, sensors and relays: only to record assumptions. But even in the simple sluice gate example of [HJJ03], it becomes clear that choosing the perimeter of the system is a crucial question: one can consider the physical phenomena to be controlled as the height of the gate, or the amount of water flowing; or the humidity of the soil; or even the farm profits. Each such scope results in different sorts of rely-conditions.

2 Pushing out the boundaries of the system

2.1 The gas-burner

The need to start the specification phase without considering the digital system can be illustrated by examining the gas-burner example used in [HRR91]. The (interesting) physical components of the gas-burner system are:

– a processor to run the control software
– a heat request interface
– a flame sensor
– a gas valve
– an ignition transformer

The requirements, taken verbatim from [HRR91], are:

1. In order to ensure safety the gas concentration in the environment must at all time be kept below a certain threshold

2. The gas-burner should burn when heat request is on, provided the gas ignites and burns without faults
3. The gas-burner should not burn when heat request is off

And three assumptions, also verbatim from [HRR91], are given:

1. When no gas is released, the flame is extinguished after at most 0.1 seconds
2. Gas cannot ignite unless the ignition transformer is operating
3. The gas concentration will stay below the critical threshold if gas never leaks for more than 4 seconds in any period of at most 30 seconds

These requirements and assumptions, on their own, give a very sparse description of what the system is supposed to be doing. Moreover, the description hides a number of assumptions which could, on the one hand, make deployment dangerous and, on the other hand, make the specification arbitrary. The referenced paper gives the first step in formalising requirements as constructing a formal model, and defines five state variables in the digital system. They are $Heatreq$, $Flame$, Gas, $Ignition$, and $Conc$. The first four are boolean-valued, and the final one is a real-valued percentage.

Nothing in that specification constrains the use of those variables, and their relationship to the physical system is left undefined. These relationships are critical: should those variables be used as sensors, so that their value is relied upon to reflect the physical world, or are they used as a channel to send commands to the physical components of the system?

The $Heatreq$ and $Flame$ variables appear to be inputs — $Heatreq$ is the input that tells the gas-burner to turn on, and $Flame$ appears to be tied to the flame sensor component in the physical system. The $Conc$ variable, used to denote the relative gas concentration around the burner, is most likely a "ghost" variable, as the physical system has no sensor to measure gas concentration. The Gas and $Ignition$ variables must then be outputs from the system, used to control the gas valve and ignition transformer respectively.

2.2 Extending the system boundaries

What is the actual purpose of the gas-burner? The specification as developed gives the impression that the purpose is to burn gas — when the $Heatreq$ signal is on — given certain time-related constraints.

Moving the boundary outwards from that, one could say that a more accurate description of the purpose of the gas-burner is to burn gas safely. The adjective "safely" is used informally here and simply means that no explosions occur and nobody is asphyxiated or intoxicated from high concentrations of gas in the environment.

Pushing the boundary of the system out further, the purpose of the gas-burner is probably to generate heat. Perhaps this is obvious; after all, one of the signals in the referenced model is called $Heatreq$. However, that merely prompts us to ask about the precise relationship between the $Heatreq$ signal and the operation of the gas-burner. Even at this level we do not know what it is that we are trying to heat, that is, what the use of the gas-burner is.

2.3 Back to the example

One of the first things to do is look at the real requirements of the system. If we take the purpose of the system as simply to generate heat, we can quickly come up with some general requirements.

The machine's behaviour, during "normal" operation, would have requirements like:

– If *Heatreq* signal comes on at some point in time means that the gas-burner will start to generate heat soon after.
– When the gas-burner is generating heat the *Heatreq* signal must be on and must have come on in the relatively recent past.
– When the *Heatreq* signal turns off then the gas-burner will stop generating heat soon after.

These requirements would be based on assumptions like:

– A flame in the gas burner generates heat.
– The presence of gas and a spark will cause a flame.
– Gas is present if the gas valve is turned on.
– The ignition transformer generates sparks.
– The gas-burner can sense the state of the *Heatreq* signal in a timely manner.

The assumptions tend to be very simple, but each can be easily formalized if necessary. Note that the sample requirements here are not intended to cover unusual situation — they are intended for a perfect environment.

The requirements for the machine when faced with an imperfect environment could include:

– The machine does not cause explosions.
– The machine does not cause toxic concentrations of gas in the environment.

This requirement forces us to consider assumptions like:

– A large concentration of gas can cause an explosion.
– Small concentrations of gas can not cause an explosion.
– The environment cannot change in such a way so that the maximum safe concentration of gas is less than some specific amount.
– The concentration of gas in the environment increases when the gas is on without a flame.
– The concentration of gas in the environment cannot increase when the gas is off.
– The environment causes concentrations of gas to dissipate over time.
– The machine will only have to deal with a single type of gas.
– The characteristics of the gas — volatility, ignition temperature, etc. — are constant during operation.
– The ignition transformer is the only source of sparks.
– There is no other source of gas in the environment other than the gas-burner.

– The environment does not actively inhibit gas-burning, but it is possible for the environment to extinguish the flame even while the gas is on.
– The gas valve cannot fail to close.
– It is also assumed that the rate of flow of gas is constant, or has a constant maximum. This is dependent on nozzle size, gas pressure and so on.

All of these assumptions are important, though this is not intended to be an exhaustive list. While many may seem trivial, violating any of them can cause a situation where the machine cannot meet its guarantee-conditions, and thus — potentially fatally — fail to meet the requirements.

From all of the requirements and assumptions above we can consider the behaviour of our machine. The observable behaviour is given through the use of guarantee-conditions, i.e.:

– The ignition transformer generates a spark after gas is turned on.
– The time between turning the gas on and the ignition transformer generating a spark is much less than the amount of time it would take for the concentration of gas in the environment to exceed a certain threshold.
– If the gas fails to ignite then the gas will be turned off, and will not be turned back on for a period of time.

Among others, there would also be guarantee-conditions that covered the specific relationship between the $Heatreq$ signal and the actions of the gas-burner.

To put the structure of the overall system into perspective it is useful to create a problem diagram of the sort described in Jackson's book [Jac00]. The diagram then acts as an aid when identifying the assumptions and possible sources of interference about which the specification needs to be concerned. Figure 2 gives a possible problem diagram from the gas-burner.

The "Control Machine" domain is the digital system whose specification we want to determine and the "Gas-Burner" domain is the physical gas-burner. The "Environment" domain represents the environment in which the gas-burner is placed. The oval labelled "Requirements" shows the relationship between the three domains it connects and shows that the behaviour of the gas-burner is what is being constrained.

The last domain, "Control Signals", was left aside as its presence while working on the diagram highlighted an important omission from the original description in [HRR91]: precisely what is controlling the $Heatreq$ signal? Even more than just that single example, the diagram also makes the possibility of change in the environment more explicit and shows — by omission — that it is strictly not possible for the machine to inspect the concentration of gas in the environment. The combination of rely-conditions and problem diagrams provide a very good means of identifying the properties — assumed or otherwise — of the overall system.

The problem diagram has the useful effect of giving a visual representation of the possible sources of interference that need to be recorded by rely-conditions. Every variable shared between domains in the diagram will, at the very least,

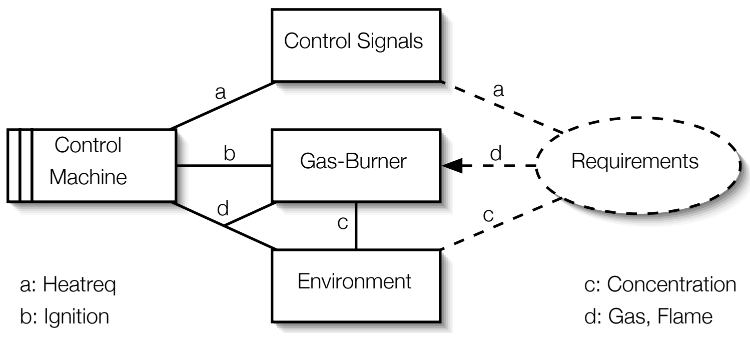


Fig. 2. Problem diagram for the gas-burner

need a rely-condition that describes the behaviour we assume it will have. Furthermore, we will also need a rely-condition for every situation where two (or more) variables have some relationship in their values.

Assumptions like the characteristics of the gas and nozzle — volatility, rate of flow, and so on — can be coded as rely-conditions fairly directly. This can even allow for some of the rely-conditions to be derived more-or-less automatically, rather than written down without any context.

The rely-conditions and the properties of the overall system are used to justify the set of guarantee-conditions that fulfill the requirements. The combination of rely- and guarantee-conditions, matched against the requirements, form the basis on which the user makes the decision as to whether or not the machine's behaviour is suitable.

Despite the linear presentation here, the construction of requirements, rely- and guarantee-conditions, problem diagrams, and the identification of assumptions is not done in a linear fashion. All of these specific tools should be used to influence the others.

3 Avoiding confusion between assumptions and requirements

The message of the general method (Section 1) as exemplified by the previous section applies to all examples: clarify the requirement in the real world before trying to specify the software which sits within the system. This process naturally identifies assumptions about the physical components which can be recorded (as in [HJJ03]) as rely-conditions.

As an indication that there is another danger of focussing too early on the computer system, we identify some reservations about one of the many specifications of the "Production Cell" example. This interesting problem is explored using many different approaches in [LL95]. The specification which we investigate is [MC94] (which is the journal version the paper by MacDonald and Carrington in [LL95]).

For the purposes of this workshop version of the paper, we assume that the reader is familiar with the overall problem.[1]

3.1 Normal operation

- Section 2 of [MC94] contains an argument for the assumption that the Feed Belt can contain only one metal block at a time (and a discussion of how changing this assumption would change the model). This is not presented as an assumption in the description; it becomes hidden in the state abstraction for $Component\_Loaded$.
- There are several places (e.g. Sections 3, 4.1, 4.2, 5.1 of [MC94]) where assumptions are made on the initial state of the system.
- A specific concern about Z is that it does not specifically identify pre-conditions of operations; this raises the question whether this decision contributes to the confusions (e.g. Section 3.2 of [MC94])
- It can be concluded from the specifications of *Extend* and *Retract* (in Section 4.1 of [MC94]) that these operations are not allowed to change *load_pos* or *unload_pos* but it is unclear whether this is an assumption on the equipment or a requirement on the code.
- Similarly, the specifications of *Load* and *Unload* (in Section 4.1 of [MC94]) indicate in their predicates that these operations are only allowed in certain positions; in this case (unlike the previous one) it might well be a requirement on the code.
- Section 4.3 of [MC94] has requirements about not rotating the robot if either arm is extended but it is left to guesswork as to whether this is an assumption on the equipment or a requirement on the code.
- Section 4.2 of [MC94] makes statements about "the press must be empty" without clarifying whose responsibility it is to achieve this situation.
- Similarly for unloading requiring that there is something to unload.
- Section 7 of [MC94] states that "the pre-condition[2] ensures there is no collision between the loaded robot and the elevating rotary table"!
- usw. usw.

4 Further work

The most obvious immediate objective is to completely formalise the examples discussed in this paper in Hayes-Mahoney logic [MH91]. Tackling these and similar further examples will inevitably refine the method described in [HJJ03]. Less immediately, further work includes creating a library of examples — including the two given here — to create a body of work that can serve as a guide to practitioners. These examples would need to be fully formal, and worked out up to the point where an implementation would be designed.

[1] Very briefly, the system has a press unit to which items are transferred from a belt by a lifting device.

[2] of $Move\_ERT\_to\_Loading\_Position\_1$

In the longer term, it should be possible to use such a library of examples to generate a set of "HJJ patterns", not unlike the design patterns [GHJV95] currently used by practitioners of object-oriented development. Even if a set of pattern-like structures cannot be developed, a full set of guidelines for using this method is required.

The composition of specifications given with this method, in senses of both subproblems and whole specifications, is a problem that remains to be fully explored. The task of creating a specification for a machine's "normal" operation seems well understood, and creating the specification with weaker rely-conditions for the "abnormal" machine behaviour is equally straightforward. However, the problem of combining such specifications is a problem that demands further study.

The basic ideas involved in the Jones' rely-conditions, while good at recording interference, leave gaps when it comes to notions such as ensuring that the system can make progress. Work such as Stølen's on wait-conditions [Stø91] addresses some of these issues, and should be included in this method.

The notation given in Jackson's [Jac00] for problem diagrams needs extension to be able to directly record interference notation. The current notation does not allow for more than a single domain to control a variable. Figure 2 is less detailed than it might have been because of this.

Acknowledgments
The authors are both supported in their research by EPSRC (UK) funding for the "Dependability IRC" (DIRC — see www.dirc.org.uk) and by EU-IST STREP funding for "RODIN" (see rodin.cs.ncl.ac.uk). The many discussions about this topic with Michael A. Jackson have been a wonderful source of insight into this material.

References

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[HJJ03] I. J. Hayes, M. A. Jackson, and C. B. Jones. Determining the specification of a control system from that of its environment. In Keijiro Araki, Stefani Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer Verlag, 2003.

[HRR91] K. M. Hansen, A. P. Ravn, and H. Rischel. Specifying and verifying requirements of real-time systems. In *SIGSOFT '91: Proceedings of the conference on Software for Critical Systems*, pages 44–54, New York, NY, USA, 1991. ACM Press.

[Jac00] M. A. Jackson. *Problem Frames: Analyzing and structuring software development problems.* Addison-Wesley, 2000.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference.* PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[LL95] C. Lewerentz and T. Lindner, editors. *Formal Development of Reactive Systems - Case Study Production Cell*, volume 891 of *Lecture Notes in Computer Science*. Springer, 1995.

[MC94] A. MacDonald and D. Carrington. Z specification of the production cell. Technical Report 94-46, University of Queensland, 1994.

[MH91] B. Mahony and I. J. Hayes. Using continuous real functions to model timed histories. In P. Bailes, editor, *Engineering Safe Software*, pages 257–270. Australian Computer Society, 1991.

[Stø91] K. Stølen. An attempt to reason about shared-state concurrency in the style of VDM. In *VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*, pages 324–342, London, UK, 1991. Springer-Verlag.

FMEA-technique of Web Services Analysis and Dependability Ensuring

Anatoliy Gorbenko, Vyacheslav Kharchenko, Olga Tarasyuk

Department of Computer Systems and Networks (503)
National Aerospace University
17 Chkalov Str., Kharkiv, 61070 Ukraine
A.Gorbenko@csac.khai.edu, V.Kharchenko@khai.edu,
O.Tarasyuk@csac.khai.edu

Abstract. Dependability analysis of the Web Services (WSs), disclosure of the possible failures modes and their effects are an actual problem. In the paper the results of the Web Services dependability analysis by using standardized FMEA-technique are represented. Obtained results were used for determining the necessary means of failure effect recovery, failure prevention, fault-tolerance ensuring and fault removal.

1 Introduction

The Web Service architecture [1] based on SOAP, WSDL and UDDI specifications is rapidly becoming a de facto standard technology for organization of global distributed computing and achieving interoperability between different software applications running on a variety of platforms.

It is now extensively used in developing various critical applications such as banking, auctions, Internet shopping, hotel/car/flight/train reservation and booking, e-business, e-science, business account management. This is why analysis and ensuring dependability in this architecture is an emerging area of research and development [1–3].

The Web Service dependability consists of several constituents, first of all, availability, reliability, security, performance/responsiveness, etc. For the e-commerce, in particular, the serviceability describing the user's satisfaction and the availability of the required services are an important characteristics.

Performance and responsiveness undoubtedly are the important characteristics, but it is easy to provide them by using the parallel computing (web-clusters) and hardware upgrading (but this is outside of the scope of this report). In this paper we will focus on ensuring of Web Service reliability and availability.

To improve the Web Services dependability and ensure fault-tolerance it is necessary to analyse possible failures modes, their causes and influence on system. For that the standardized FMEA-technique [4] was used.

2 Analysis of the Web Services by Using FMEA-Technique

The FMEA (failure modes and effects analysis) is a standard formalized technique for the reliability analysis of different systems which devoted to the specification of failure modes, their sources, causes of occurrence and influence on system as a whole [4]. The use of the FMEA-technique for the Web Services analysis allows to identify the typical failures and their influence on the Web Services dependability, and also to determine necessary means for fault-tolerance and failure effect recovery. FMEA-technique may be an important part of Web Services dependability guaranteeing program.

Computer system provided some Web Service consists of hardware and specific software components (web server, application server, DBMS, application software – servlets, stored procedures and triggers) and may have different architectures (Fig. 1). These components must be taken into account during failure modes and effects analysis.

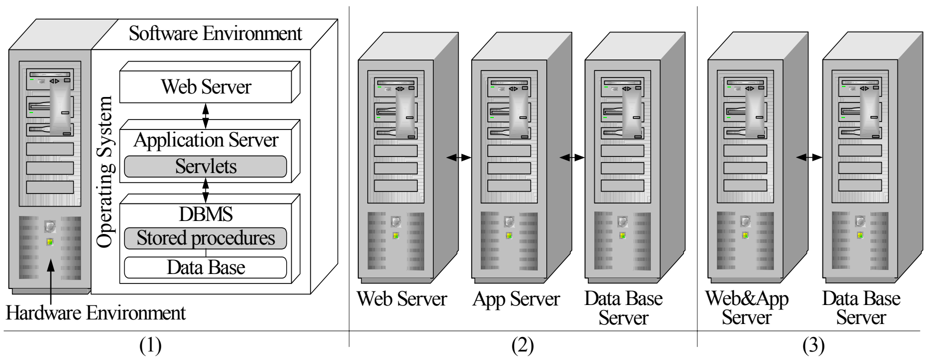


Fig. 1. Typical Web Services component architectures: (1) all components in the same computer; (2) fully separated component architecture; (3) partially separated component architecture.

The analysis of Web Services failures modes, causes and effects is obtained by using the FMEA-format (Tables 1, 2). To reduce scale of FMEA-tables we replaced repeating rows by arrows.

To identify the Web Services failures modes new failure taxonomy was proposed (Fig. 2) taking into consideration variants described in [5–8]. The proposed taxonomy classifies possible failures from the points of view of the Web Service publishers and the end-users and takes into account failure domain, failure evidence and stability of occurrence, and also its influence on system operability.

We performed analysis of failure effect on data, system components, users and Web Services as a whole. Several failures modes can lead to the prolonged or short-term service aborting that affects on users as denial of service. But some failures result in a non-evident incorrect service. For many applications (e-commerce, critical automation control, etc.) such effect is more dramatic because will entail serious consequences, financial loss and, finally, service discrediting.

Table 1. Hardware failures modes end effects analisys

Failure Domain	Stability of Occurrence	Failure Cause	Influence on Operability	Failure Evidence	Failure Effect					
					on HW	on SW	on stored data	on session data & calculation	on web service as a whole	on user
HW environment	accidental failures	1) HW deterioration; 2) pernicious external influence	termination	evident	crash	crash	corruption	data loss	service abort	deny of service
				evident	crash	suspension	–	data loss	service abort	deny of service
		non-pernicious external influence (interference)	termination	evident	hang	crash	corruption	data loss	service abort	deny of service
			interruption	evident	hang	suspension	–	data loss	service abort	deny of service
				evident	rebooting	restarting	–	data loss	service abort	deny of service
	permanent failures	design faults	–	evident	–	–	–	data/ calculation error	service exception	deny of service
				non-evident	–	–	–	data/ calculation error	–	incorrect service

Table 2. Software failures modes end effects analisys

Failure Domain		Stability of Occurrence	Failure Cause	Influence on Operability	Failure evidence	Failure Effect					
						on HW	on SW	on stored data	on session data & calculation	on web service as a whole	on user
SW environment	OS	transient failures	design fault	termination	evident	hang	crash	corruption	data loss	service abort	deny of service
	Web Server			interruption	evident	hang	OS/Servers/ DBMS/App suspension	–	data loss	service abort	deny of service
	App Server		malicious impact	interruption	evident	rebooting	restarting	–	data loss	service abort	deny of service
	DBMS	permanent failures	(hacker attack, viruses)	–	evident	–	–	–	data/ calculation error	service exception	deny of service
Application SW	Servlets		incorrect input data	–	non-evident	–	–	–	data/ calculation error	–	incorrect service
	Stored procedures & triggers										

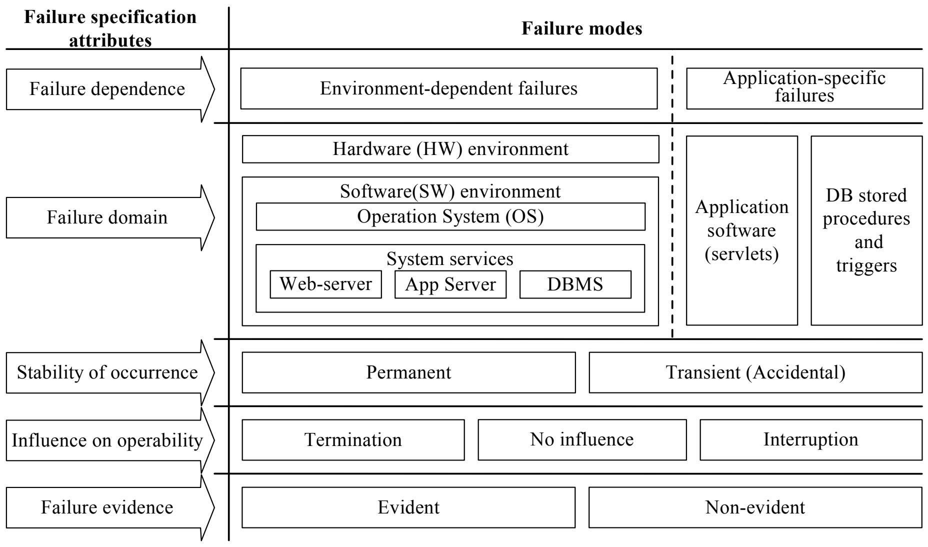


Fig. 2. Failure taxonomy

As it was inquired, the hardware design faults (faults in processors, chipsets, etc.) still remain one of the possible causes of the Web Services failures. Furthermore, a monthly Specification Update for Intel product series can contain up to several tens of errata, some of which under certain circumstances can lead to unexpected program behavior, calculation error or processor hang. However, the prevalent sources of Web Services failures are the different software components.

The reliability (probability of failure-free operation) of separated Web Services architectures presented on the Fig. 1 (2, 3) is less than reliability of concentrated architecture (1) because of the increase of a number of HW components that can fail. Thus, such architectures are expedient for using in cluster systems.

Performed analysis will help in defining the necessary failure recovery and fault-tolerance means for specific failure modes. Set of the fault-tolerance means depends on failure modes and causes whereas the required failure recovery means depends on failure effect on system and its components. Failures severities can be defined by their evidence and influence on system operability.

3 Ensuring Web Services Dependability and Fault-Tolerance

3.1 Failure effect recovery

Common means of the failure effect recovery for Web Services include: 1) replacement of crashed hardware components; 2) reinstall of crashed software components; 3) data recovery; 4) system rebooting or restarting of the particular software services.

To achieve better availability system rebooting and restarting of the particular software services and applications must be performed in automatic mode with the help of hardware or software implemented watch-dog timer. Besides, it is preferentially to have secure way for remote system rebooting by administrator.

It is very important to perform regular data backup for success data recovery.

3.2 Failure prevention

Fault prevention is attained first of all by quality control techniques employed during the design and manufacturing of hardware and software [5]. However, most of the hardware and software Web Services components are the COTS- (commercial of the shelf) components developed by third parties.

Hence, service publisher has limited means for failure effect prevention:

- quality control techniques employed during the design of the own developed application software;
- procedures for input parameter checking;
- rigorous procedures for system maintenance and administration;
- firewalls, security guards and scanners to prevent malicious failures.

Besides, to prevent transient failures and performance reducing caused by software rejuvenation can be used techniques based on forced restarting/reinitialization of the software components [9].

3.3 Fault-tolerance

The development of fault tolerant techniques for the Web Services has been an active area of research over the last couple of years. The backward (based on rolling system components back to the previous correct state) and forward (which involves transforming the system components into any correct state) error recovery for the web on the basis of an application-specific exception handling is discussed in [10].

More generally, high dependability and fault-tolerance of the Web Service is ensured by using different kinds of redundancy and diversity at the different levels of the system structure (Fig. 3). HW redundancy may be partial (redundancy of processors, hard discs – RAID, network adapters, etc.) as well as complete with replication or diversification of SW. Complete HW and SW redundancy is a foundation of cluster architectures and provides better performance and dependability.

Diversity is used usually to tolerate software or hardware failures caused by design faults. But for tolerating transient failures a simple replication of SW environment with HW redundancy may be a sufficient means because of the individual behavior even of two replicated SW environment. To tolerate non-evident failures the voting scheme must be used.

The 72-87% of the faults in open-source software are independent of the operating environment (i.e. faults in application software) and are hence permanent [6]. Half of the remaining faults is environment depended and permanent. And only 5-14% of the faults are environment depended caused by transient conditions. Hence, diversity is the most efficient method of fault-tolerance provision. It can be used for HW platform, OS, web and application servers, DBMS and, finally, for application software both separately and in many various combinations.

However diversity can worse the intrusion-tolerance and Web Service security (confidentiality and integrity) because it opens new potential ways for malicious intrusions. At the same time diversity brings additional protection against DoS attacks.

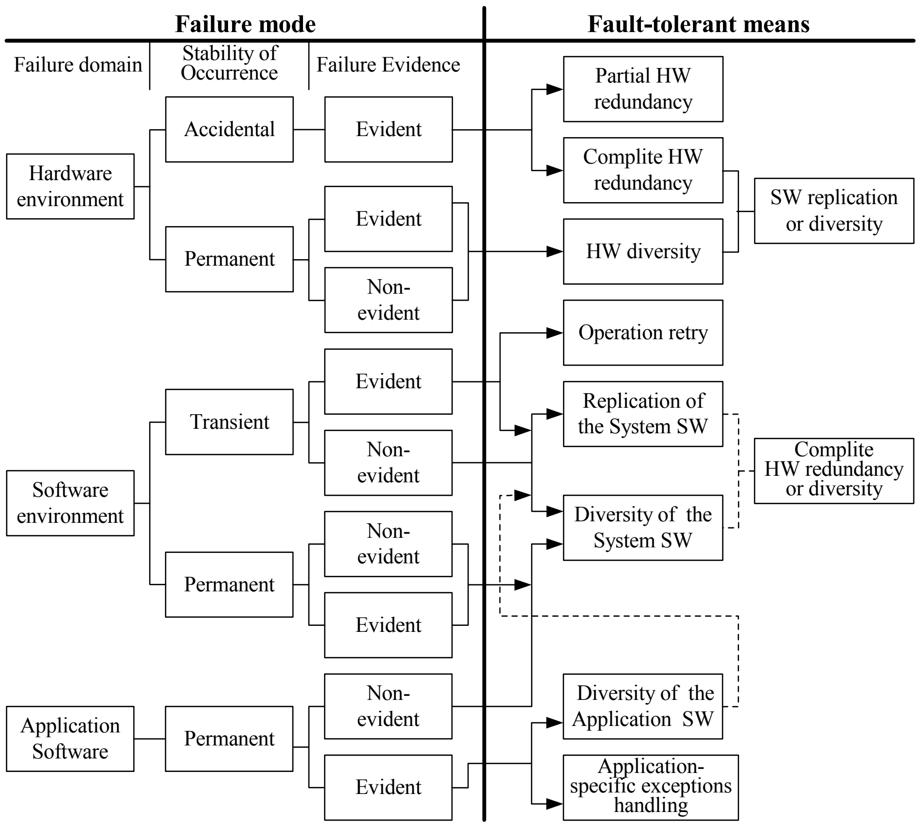


Fig. 3. Means for Web Services fault-tolerance

3.4 Fault removal

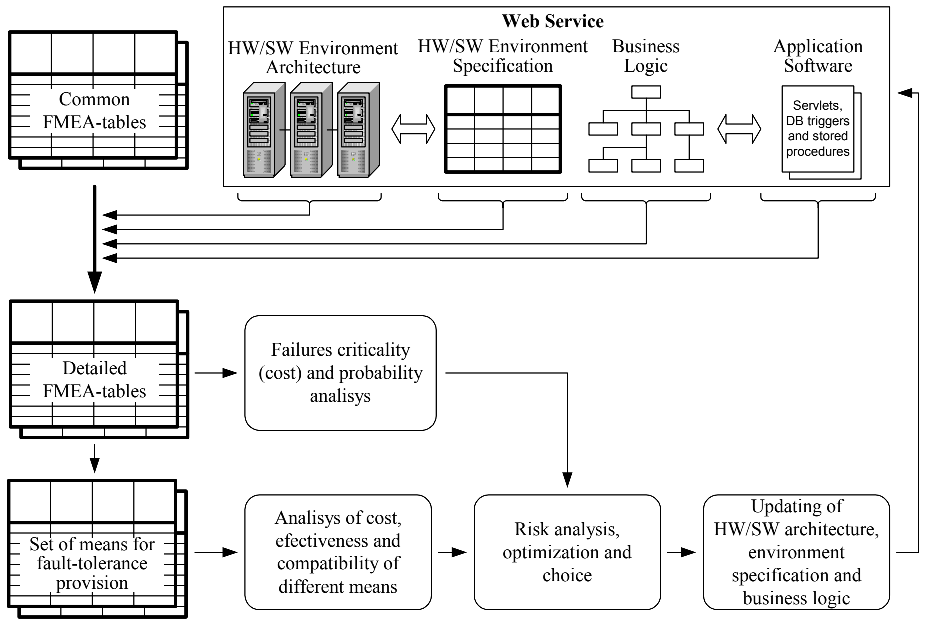
Fault removal of the Web Services based, first of all, on the systematic applying of the updates and patches for hardware (microcode updates) and software developed by third parties (OS, drivers, web and app servers, DBMS).

Fault removal from the own developed application software is performed both during the development phase and the maintenance.

4 Dependable Web Services Development and Deployment

4.1 Using FMEA-technique for Dependable Web Services Development

To develop and deploy dependable Web Services the common FMEA-tables (see Tables 1-2) describing hardware and software failures modes and effects must be concretized taking into account actual hardware/software architecture of particular Web Service (Fig. 4).



Web Service
HW/SW Environment Architecture
HW/SW Environment Specification
Business Logic
Application Software
Servlets, DB triggers and stored procedures
Common FMEA-tables
Detailed FMEA-tables
Failures criticality (cost) and probability analisys
Set of means for fault-tolerance provision
Analisys of cost, efectiveness and compatibility of different means
Risk analysis, optimization and choice
Updating of HW/SW architecture, environment specification and business logic

Fig. 4. Using FMEA-technique for Dependable Web Services Development

The two different development strategies are possible. For Web Services of business-critical applications (for example, e-commerce) it is necessary as a rule to provide the required dependability at the minimum costs. For Web Services of commercial applications it is important to provide the maximum dependability at the limited costs. These goals can be achieved by solving optimization problem taking into account failures criticality, probability of occurrence and cost of fault-tolerance means, their effectiveness and failures coverage. As a result the Web Service must be updated using chosen fault-tolerance means.

4.2 The principles of Dependable and Secure Web Services Deployment

The Web Services fault and intrusion tolerance, security and dependability as a whole can be improved by using following principles.

1. Defense in depth and diversity (D&D). This principle provides the using of diversity at the different levels of the Web Service architecture (HW platform, OS, System SW, etc.) and also joint usage of existed security and fault-tolerance facilities. Here, the compatibility between different facilities and diversity modes must be taken into account. To solve this problem the multilevel diversification graph can be used [11]. The number of graph levels will be equal to the number of diversified components of the Web Services whereas the number of nodes at the each level will be equal to the number of existed diverse elements.
2. Adaptability and update (A&U). The essence of this principle is in the dynamic changing of Web Service architecture and diversity modes according to observed failures and intrusions. For that the intellectual monitoring means can be used in the system for the detection of the failures and intrusions, their analysis and the choice of the better Web Service configuration. These means can include external alarm services to notify about recent Internet security vulnerabilities, novel viruses and to distribute security updates and patches.

The D&D and A&U principle are corresponding to the DIT (Dependable Intrusion Tolerance) architecture described in [12].

5 Conclusions

Publishers of Web Services have a limited possibility for fault prevention and fault removal of the most components of Web Services, developed by third parties. Thus, redundancy in combination with diversity is one of the basic means of dependability ensuring and tolerance provision to the majority failure modes. But using diversity in Web Service architecture requires detailed researches and addition solutions because it can lead to the addition security violations.

Cluster architecture improves availability of Web Services. The additional adaptive reliable algorithms and means of voting and failures diagnosis must be implemented for the ensuring tolerance to the non-evident failures and prevention of losses of the processed (in-service) requests.

The FMEA is an effective technique, which can be used for the application of a specific dependability analysis of Web Services, especially of composite WSs. Fulfilled analysis can be extended by taking into account the lacks of required resources or services and service unavailability due to network failures. Besides, the critical analysis of different failures modes can be performed.

FMEA-tables may be dynamically updated during Web Service operation. It allows (jointly with implementation of D&D and A&U principles) to increase the effectiveness of the used means of dependability ensuring.

References

1. W3C Working Group.: Web Services Architecture. http://www.w3.org/TR/ws-arch/ (2004)
2. Ferguson, D.F., Storey, T., Lovering, B., Shewchuk, J.: Secure, Reliable, Transacted Web Services: Architecture and Composition. Microsoft and IBM Technical Report. http://www-106.ibm.com/developerworks/webservices/library/ws-securtrans (2003)
3. Tartanoglu, F., Issarny, V., Romanovsky, A., Levy, N.: Dependability in the Web Service Architecture. In: Architecting Dependable Systems. Springer-Verlag (2003) 89–108.
4. IEC 812. Analysis Techniques for System Reliability – Procedure for Failure Modes and Effects Analysis (FMEA). International Electrotechnical Commission, Geneva (1985)
5. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing, Vol. 1(1) (2004) 11–33
6. Chandra, S., Chen, P. M.: Whither Generic Recovery From Application Faults? A Fault Study using Open-Source Software. Proc. Int. Conf. on Dependable Systems and Networks (2000) 97–106
7. Deswarte, Y., Kanoun, K., Laprie, J.-C.: Diversity against Accidental and Deliberate Faults. Proc. of Computer Security, Dependability, and Assurance (SCDA): From Needs to Solutions, York, England (1998) 171–181
8. Gorbenko, A., Kharchenko, V., Popov, P., Romanovsky, A., Boyarchuk, A.: Development of Dependable Web Services out of Undependable Web Components. CS-TR 863. School of Computing Science, University of Newcastle upon Tyne, UK (2004).
9. Vaidyanathan, K., Harper, R. et al.: Analysis and Implementation of Software Rejuvenation in Cluster Systems. Proc. Joint Intl. Conf. Measurement and Modeling of Computer Systems, ACM Sigmetrics and IFIP WG 7.3, Cambridge (2001) 62–71
10. Tartanoglu, F., Issarny, V., Romanovsky, A., Levy, N.: Coordinated Forward Error Recovery for Composite Web Services. Proc. 22nd IEEE Symposium on Reliable Distributed Systems (2003)
11. Kharchenko, V.: Multiversion Systems: Models, Reliability, Design Technologies. Proc. 10th European Conference on Safety and Reliability, Munich, Germany, Vol.1(1999) 73–77
12. Valdes, A., Almgren, M., Cheung, S., Deswarte, Y. et al.: An Architecture for an Adaptive Intrusion-Tolerant Server. Proc. 10th Int. Workshop on Security Protocols (2002), Lecture Notes in Computer Science, 2845 ed, Cambridge, UK: Springer (2004) 158–178

Modelling Fault Tolerance of Transient Faults

Dubravka Ilic and Elena Troubitsyna

Åbo Akademi, TUCS, Department of Computer Science,
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
{Dubravka.Ilic, Elena.Troubitsyna}@abo.fi

Abstract. In this paper we focus on analysis of transient physical faults and designing mechanisms to tolerate them. Transient faults are temporal faults that appear for some time and might disappear and reappear later. They are common in control systems. However transient fault appearing even for a short time might result in a system error. Hence fault tolerance mechanisms for detecting and recovering from temporal faults are of great importance in the design of control systems. Often the system module which detects errors and performs error recovery is called a Failure Management System. Its purpose is to prevent the propagation of errors in the system. In this paper we propose a formal approach to specifying the Failure Management System in the B Method. We focus on deriving a general specification and development pattern for Failure Management Systems for tolerating transient faults.

1 Introduction

Nowadays software-intensive control systems are in heart of many safety-critical applications. Hence dependability of such systems is a great concern. While designing controlling software for such systems we should ensure that it is able not only to detect errors in system functioning but also to confine the damage and perform error recovery. In this paper we focus on designing controllers able to withstand transient physical faults of the system components [9]. Transient faults are temporal defects within the system. We focus on analysis and design of a special subsystem of control systems – a Failure Management System (further referred to as FMS) – which performs error detection, damage confinement and error recovery. The FMS is a subsystem of the embedded control system responsible for providing the controller with the error free inputs obtained from the environment. Since controller is relying only on the input from FMS, it is important to ensure its correctness.

Design of the FMS is particularly difficult since often requirements changes are introduced at the late stages of the development cycle. These changes are unavoidable since many requirements result from empirical performance studies executed under failure conditions. To overcome this difficulty we propose a formal pattern for specifying fault tolerance mechanism in the FMS. The contribution of our work is in verifying the suggested pattern rather then a particular specification. The proposed pattern can be reused in the product line development and hence its correctness is crucial.

We demonstrate how to develop the FMS by stepwise refinement in the B Method [3]. Our approach is validated by a realistic case study conducted within EU project RODIN [7].

2 Fault tolerance mechanism in FMS

Failure Management System (FMS) [2] is a part of the embedded control system responsible for managing failures of the system inputs as shown on Figure 1.

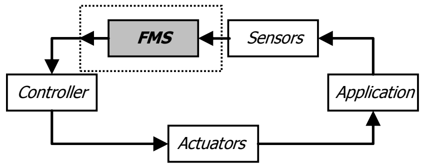


Figure 1. Place of the FMS in an embedded control system

The main role of FMS is to supply the controller of the system with the error free inputs from the system environment.

All inputs supplied to the FMS are analysed. The analysis of each input results in invocation of the corresponding remedial action. There are three categories of remedial actions: healthy, temporary or confirmation actions. If an input is considered to be error free, it is forwarded unchanged to the controller. This is a *healthy* system action. If an error is detected, the input gets suspected and the FMS decides on error recovery. The aim of FMS is to give error free output even when input is in error, i.e., during recovery phase. Hence, when the input is suspected, the system sends the last good value of the input as the error free output toward the controller. This is a *temporary* system action. In the recovery phase the input can get recovered during certain number of operating cycles. If the input fails to recover, the *confirmation* action is triggered and the system becomes frozen.

In Figure 2 we illustrate the behaviour of FMS over one analogue input.

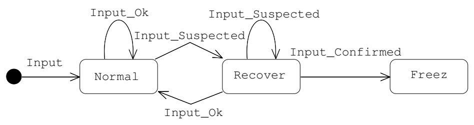


Figure 2. Specification of the FMS behaviour

A general description of FMS behaviour is as follows: after getting the input from the environment through the system sensors, the FMS determines whether the input is in error or error free. If the input is error free, the FMS applies healthy remedial action. If it is in error, it is classified as suspected and the system initiates recovery phase. When the recovery starts, a counting mechanism responsible for ensuring the recovery termination is triggered. If after recovery the input is still suspected, the confirmation action is applied, i.e., the input is confirmed failed and the system freezes. Otherwise, the system considers the input again as error free, applies the healthy action and continues the operation without any interruption.

The general description of FMS behaviour lacks, however details about the error detection.

When an input is received by FMS, FMS performs certain tests on the inputs to determine its status: in error or error free. We differentiate between the individual and collective tests. Individual tests (e.g., `Test1` and `Test2` in Figure 3) are obligatory for each input and they determine the preliminary abnormality in the input. When triggered, individual tests run solely based on the input reading from the sensor. We use two kinds of individual tests: the magnitude test and the rate test. In the magnitude test the input is compared against some predefined limit (bound) and if exceeds, it is considered in error. The rate test is detecting erroneous input while comparing the change of the input readings in consecutive cycles. Namely, the current value of the input is compared against the previous input value and if some predefined limit is exceeded, the input is considered in error. It is obvious that both tests have some preconfigurations expressed through the predefined limits which allow dynamic test changes as appropriate.

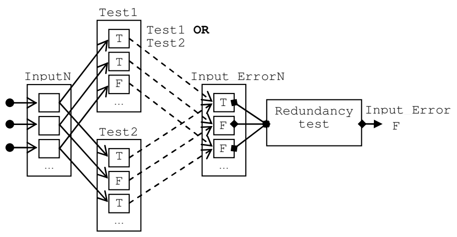


Figure 3. Introducing error detection

The error detection for multiple sensors (`InputN` in Figure 3) implies first the application of individual tests and then, when these tests are passed, the collective test is applied. The collective test is commonly a redundancy test. It is applied on the group of multiple sensor inputs. As presented on the Figure 3, redundancy test takes the detected multiple inputs (`Input_ErrorN`) and based on their values (`TRUE` or `FALSE`) votes for the input status (`Input_Error`). This status becomes `TRUE` (i.e., the input is considered in error) if there are more erroneous inputs for the multiple sensor readings then error free ones. When the input status is finally detected, FMS proceeds with the corresponding remedial actions.

Before presenting our formal pattern for handling fault tolerance in FMS, we give the short introduction to the B Method.

3 Formal system modelling in the B Method

In this paper we have chosen the B Method [3] as our formal modelling framework. The B Method is an approach for the industrial development of correct software. The method has been successfully used in the development of several complex real-life applications [6]. The tool support available for B, for instance - Atelier B [1], provides us with the assistance for the entire development process.

In this paper we adopt event-based approach to system modelling [4]. The events are specified as the guarded operations `SELECT cond THEN body END`. Here `cond` is a state predicate, and `body` is a B statement describing how state variables are affected by the operation. If `cond` is satisfied, the behaviour of the guarded operation corresponds to the execution of its `body`. If `cond` is false at the current state then the operation is disabled, i.e., cannot be executed. Event-based modelling is especially suitable for describing reactive systems. Then `SELECT` operation describes the reaction of the system when particular event occurs.

For describing the computation in operations we used following B statements:

Statement	Informal meaning
`X := e`	Assignment
`IF P THEN S1 ELSE S2 END`	If `P` is true then execute `S1`, otherwise `S2`
`S1 || S2`	Parallel execution of `S1` and `S2`
`X :: T`	Nondeterministic assignment – assigns variable `x` arbitrary value from given set `T`

The last statement allows for abstract modelling and hence, postponing implementation decisions till later development stages.

The development methodology adopted by B is based on stepwise refinement [8]. While developing a system by refinement, we start from an abstract formal specification and transform it gradually into an implementable program by a number of correctness preserving steps, called refinements. In the refinement process we reduce non-determinism of the original specification and eventually arrive at deterministic implementable specification.

The result of a refinement step in B is a machine called `REFINEMENT`. Its structure coincides with the structure of the abstract machine. However, refined machine should contain an additional clause `REFINES` which defines the machine refined by the current specification. Besides definitions of variable types, the invariant of the refinement machine should contain the refinement relation. This is a predicate which describes the connection between state spaces of more abstract and refined machines.

To ensure correctness we should verify that initialization and each operation preserve the invariant. Verification can be completely automatic or user-assisted.

Next we demonstrate how to formally specify failure management system described in Section 2.

4 Formal development of FMS

4.1 Specifying the failure management system

Control systems are usually cyclic, i.e., their behaviour is essentially an interleaving between the environment stimuli and controller reaction on these stimuli. The controller reaction depends on whether the FMS has detected error in the obtained input. Hence, it is natural to consider the behaviour of FMS in the context of the overall system.

The FMS gets certain inputs from the environment, applies specific detection mechanisms and depending on the detection results produces output to the controller or freezes the whole system. Inputs that FMS receives from the environment are inputs from various sensors. In this paper we consider only analogue sensors. In absence of errors the output from the FMS is the actual input to the controller. However, if error is detected the FMS should try to tolerate it and produce the error free output or to stop the system without producing any output at all.

In our abstract specification given in Figure 5, for modelling fault tolerance on given input we used different variables. The variable `FMS_State` defines the phases of control cycle execution. Its values are as follows: `env` – obtaining inputs from the environment, `det` – detecting erroneous inputs, `act` – changing the system operating mode, `rcv` – recovering of the erroneous input, `out` – supplying the output of the FMS to the controller, `stop` – freezing the system. The variable `FMS_State` models the evolution of system behaviour in the operating cycle. At the end of the operating cycle the system finally reaches either the terminating (freezing) state or produces the error free output. After the error free output was produced, the operating cycle starts again. Hence, the behaviour of the FMS can be described as in Figure 4.

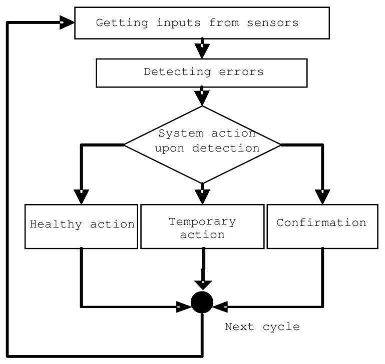


Figure 4. Behaviour of the FMS

Since the controller relies only on the input from the FMS, we should guarantee that it obtains the error free output from the FMS. Hence, our safety invariant expresses this: whenever the input is confirmed failed, the FMS output is not produced (i.e., `Input_Status=confirmed => FMS_State=stop`) and, whenever the input is confirmed `ok`, the output should have the same value as input or be different if the input is `suspected` (i.e., `(Input_Status=ok => Output=Input) & (Input_Status=suspected => Output/=Input))`.

Although the abstract specification of FMS is highly abstract it anyway specifies the fault tolerance mechanism allowing us to ensure the desired behaviour of the system. In this abstract specification the input values produced by the environment are modelled nondeterministically. After getting the inputs, FMS performs detection on inputs to determine if they are in error or error free. This is modelled in the *Detection* operation of the FMS machine as a nondeterministic assignment of some boolean value (TRUE or FALSE) to the variable modelling input state (i.e., Input_Error :: BOOL). After the input state is detected, FMS triggers the healthy action if the input is error free. If the input is in error, FMS initiates temporary action, i.e., error recovery.

```
MACHINE
    FMS
SEES
    Global
VARIABLES
    Input, Input_Error, FMS_State, cc, num
INVARIANT
    Input : T_INPUT &     /*actual input to the FMS*/
    Input_Error : BOOL &  /*variable modelling input
                            status*/
    FMS_State : STATES &  /*variable modelling system
                            state*/
    cc : NAT &            /*cc and num are counters*/
    num : NAT &
    <safety invariant>
INITIALISATION
    FMS_State :=env || cc:=0 || num:=0
OPERATIONS

    Environment=
    SELECT <the system is functioning normally>
    THEN
           <nondeterministically choose some input> ||
           FMS_State:=det
    END;

    Detection=
    SELECT <the system is in the detection state>
    THEN
           Input_Error :: BOOL || FMS_State:=act
    END;

    Action=
    SELECT <the input is not in error>
    THEN
           <healthy action > || FMS_State:=out
    WHEN
           <the input is in error and the
            error is just discovered>
    THEN
           <input is marked as suspected> ||
           cc:=cc+xx || num:=num+1 || FMS_State:=rcv
    WHEN
           <the input is not in error but it is already
            marked suspected>
    THEN
           <input stays suspected> ||
           cc:=cc-yy || num:=num+1 || FMS_State:=rcv
    WHEN
           <the input is in error and it is already
            marked suspected>
    THEN
           <input stays suspected> ||
           cc:=cc+xx || num:=num+1 || FMS_State:=rcv
    END;

    Return=
    SELECT <healthy action>
    THEN
           <input is passed to the output> ||
           FMS_State:=env
    WHEN
           <temporary action>
    THEN
           <output is assigned the last good
            value of the input> || FMS_State:=env
    END;

    Recovering=
    SELECT <input is suspected> & (num>=Limit or cc>=zz)
    THEN
           <input confirmed failed> || FMS_State:=stop
    WHEN
           <input is suspected> & num<Limit & cc=0
    THEN
           <input has recovered> || FMS_State:=out
    WHEN
           <input is suspected> & num<Limit & cc/=0 &
           cc<zz
    THEN
           FMS_State:=env
    END;

    Stopping=
    SELECT FMS_State=stop
    THEN
           skip
    END
END
```

Figure 5. Excerpt from the abstract FMS specification

Error recovery is modelled by introducing the two counters: cc and num. At the beginning of the operating cycle, both counters are set to zero and their values are changed only in the recovery phase. The first counter cc counts inputs which are in error. While the system is in the recovery phase, every time when the obtained input is found in error, the system sets as the output the last good value of the input and the counter cc is incremented by some given value xx. However if the input is error free, the cc is decremented by the given value yy. In each operating cycle system is setting some values for the counter cc either by decrementing or incrementing it. If at one point the value of the cc exceeds some predefined limit zz the counting stops and the system confirms the input failure by terminating the operation and freezing the system. Since each erroneous input increments the value of cc and each error free input decrements it, eventually the counter cc is set to zero. This is possible if eventually the FMS starts to receive error free inputs. If cc reaches zero the input is considered to be recovered and the system returns to normal functioning initializing cc to zero and making it thus ready for the next recovering cycle. The way cc reaches zero or exceeds the limit zz is determined via setting the parameters xx, yy and zz. These parameters are set by observing the real performance of the failure. By setting the value of xx higher then the value of yy, the counter cc is going to yield the limit zz faster. However, such a specification is insufficient for guaranteeing termination of recovery. Observe that the input may vary in such a way that the counter cc is practically oscillating between some values but never reaching the limit zz or zero. Hence, we introduce the second counter num which is counting each recovering cycle. When some allowed limit for num is exceeded, the recovery terminates and if cc is different than zero the input is confirmed failed.

Our initial specification completely describes the intended behaviour of the FMS but leaves the mechanism of detecting errors in input unspecified. Next, we demonstrate how to obtain the detailed specification of error detection in the refinement process.

4.2 Refining error detection in FMS

Since we observe multiple sensors the refinement of the FMS starts with replacing the Input variable with the InputN variable modelling the sequence of input values received by the FMS as N sensor readings, instead of only one sensor reading. The nondeterministic assignment of value to the variable Input_Error in the *Detection* operation of the abstract machine is further refined. By introducing new variable Input_ErrorN we can set the value for each particular sensor reading. Input_ErrorN is a sequence with Boolean values TRUE or FALSE. These values are determined for each multiple sensor input by running two detection tests: the magnitude test and the rate test. If the input passes the magnitude test, the value of the temporary variable Input_Error1 is set to FALSE, otherwise is TRUE (i.e., the test on this input failed). Similarly, if the input passes the rate test, the value of the temporary variable Input_Error2 is set to FALSE, otherwise TRUE.

The input is error free if none of these tests fail. Hence we define the status of the input as the disjunction of Input_Error1 and Input_Error2 and set the variable Input_ErrorN accordingly.

After setting the values of the variable Input_ErrorN in described way, we apply the redundancy test (as shown in Figure 3). We consider N sensor readings which values are stored in introduced variable InputN. Moreover, our assumption is that this number is odd to prevent the situation in which the number of the erroneous and error free inputs is the same. The status of each one of the N sensor inputs is recorded in the variable Input_ErrorN. The redundancy test performs majority voting. It means that if there are more values TRUE in the Input_ErrorN sequence, the whole input is considered failed, otherwise it is error free. After the status of the input is detected, FMS makes a decision how to proceed with handling it, i.e., which action it is going to apply as specified in the abstract specification.

The essence of our refinement step is to introduce modelling of the N sensor inputs instead of only one and replace the nondeterministic assignment to the variable Input_Error with deterministic error detection. The refinement relation for this step is as follows:

```
                    (Input_Error=TRUE =>
  (card(Input_ErrorN|>{TRUE}) > card(Input_ErrorN|>{FALSE})))
```

The above refinement relation establishes connection between the abstract variable `Input_Error` and the concrete variable `Input_ErrorN`. Namely, if the value of `Input_ErrorN` is such that the number of error free inputs is smaller then the number of erroneous inputs then it should correspond to the value `TRUE` of `Input_ErrorN`.

To produce the final output, FMS calculates the median value of all error free inputs and passes it as the output from the FMS.

In the Figure 6 we give the excerpt from this refinement step of the FMS with introduced error detection.

```
REFINEMENT
   FMSR1
REFINES
   FMS
SEES
   Global
VARIABLES
   InputN, Input_Error, Input_Error1, Input_Error2,
   Input_ErrorN,
   FMS_State,
   cc,num,
   Passed1, Passed2
INVARIANT
   InputN : seq(T_INPUT) & /*N sensor input reading*/
   Input_Error : BOOL &
   Input_Error1 : BOOL &   /*test results for 1 input*/
   Input_Error2 : BOOL &
   Input_ErrorN : seq(BOOL) & /*test results for
                               N sensor inputs*/
   FMS_State : STATES &
   cc : NAT & num : NAT &
   Passed1 : BOOL &           /*variables for modeling
                               test application*/
   Passed2 : BOOL &
   <safety and gluing invariants>
INITIALISATION
   InputN := [] || Input_Error := FALSE ||
   Input_Error1 := FALSE || Input_Error2 := FALSE ||
   Input_ErrorN := [] ||
   FMS_State := env ||
   cc := 0 || num:=0 ||
   Passed1 := FALSE || Passed2 := FALSE

OPERATIONS

   <obtaining the input from the environment>

   Detection=
   SELECT <magnitude test not passed yet>
   THEN
       IF
          <the input is in defined low and high limits>
       THEN
          Input_Error1:=FALSE
       ELSE
          Input_Error1:=TRUE
       END ||
       Passed1:=TRUE ||
       FMS_State:=det
       WHEN
           <rate test not passed yet>
       THEN
           IF
              <the input change exceeds the limit>
           THEN
              Input_Error2:=TRUE
           ELSE
              Input_Error2:=FALSE
           END ||
           Passed2:=TRUE ||
           FMS_State:=det
       WHEN
           <both test are passed>
       THEN
           IF
              /*simulate disjunction*/
              Input_Error1=Input_Error2 &
              Input_Error1=TRUE
           THEN
              /*record the input status*/
              Input_ErrorN:=Input_ErrorN <- TRUE
           ELSE
              Input_ErrorN:=Input_ErrorN <- FALSE
           END ||
           /*remove the detected input from further
             observation*/
           InputN:=tail(InputN) ||
           FMS_State:=det
       WHEN
           <input sequence InputN is empty>
       THEN
           /*apply the redundancy test*/
           IF
              <the number of TRUE values in Input_ErrorN
               greater then the number of FALSE values>
           THEN
               Input_Error:=TRUE
           ELSE
               Input_Error:=FALSE
           END ||
           FMS_State=act
       END;

       <system action upon detection>

   END
```

Figure 6. Excerpt from refining the error detection in FMS

5 Conclusion

In this paper we proposed a formal pattern for specifying and refining fault tolerant control systems susceptible to transient faults. We demonstrated how to ensure that safety requirement – confinement of erroneous inputs – is preserved in the entire development process. We focused on the design of subsystem of the control system – the failure management system, which enables error detection, confinement and recovery. Our approach has currently focused on considering multiple analogue sensors. We derived a general specification of the corresponding error detection mechanism which defines the appropriate tests run on the obtained inputs. We verified our pattern on a case study.

Laibinis and Troubitsyna [5] proposed a formal approach to model-driven development of fault tolerant control systems in B. However, they did not consider transient faults. Since we consider this type of faults our approach can be seen as an extension of the pattern they proposed.

More work on specifying FMS has been done by Johnson et. al [2]. However, they focused on reusability and portability of FMS modelled using UML in combination with formal methods. The error detection mechanism proposed here is based on the application of specific tests combined with the counting mechanism. Hence we focused on specifying the essence of mechanism for tolerating transient faults.

We verified our approach with the automatic tool support – Atelier B. Around 95% of all proof obligations have been proved automatically by the tool. The rest has been proved using the interactive prover. We believe that the availability of the tool supporting formal specification and verification can facilitate acceptance of our approach in industry.

In this paper we addressed a specific subset of transient faults. As a future work we are planning to enlarge this subset and derive generic patterns for specification and development of control systems tolerating them. Moreover, it would be interesting to investigate the possibility of automatic instantiation of specific requirements from which the general pattern is obtained.

Acknowledgments

This work is supported by EU funded research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

6 References

1. CClearSy, Aix-en-Provence, France. *Atelier B - User Manual*, Version 3.6, 2003.
2. I. Johnson, C. Snook, A. Edmunds and M. Butler. "Rigorous development of reusable, domain-specific components, for complex applications", In *Proceedings of 3rd International Workshop on Critical Systems Development with UML*, pages pp. 115-129, Lisbon, 2004.
3. J.-R. Abrial. *The B Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
4. J. R. Abrial. Event Driven Sequential Program Construction, 2001. http://www.atelierb.societe.com/ressources/articles/seq.pdf
5. L. Laibinis and E. Troubitsyna. "Refinement of fault tolerant control systems in B", In *Computer Safety, Reliability, and Security - Proceedings of SAFECOMP 2004* Lecture Notes in Computer Science, Num: 3219, Page(s): 254-268, Springer-Verlag, Sep, 2004.
6. *MATISSE Handbook for Correct Systems Construction*. EU-project MATISSE: Methodologie and Technologies for Industrial Strength Systems Engineering, IST-199-11345, 2003. http://www.esil.univ-mrs.fr/~spc/matisse/Handbook
7. RODIN - Rigorous Open Development Environment for Complex Systems, Project Number: IST 2004-511599, http://rodin-b-sharp.sourceforge.net
8. R. J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.
9. Storey N. *Safety-critical computer systems.* Addison-Wesley, 1996.

Application of Event B to Global Causal Ordering for Fault Tolerant Transactions

Divakar Yadav* and Michael Butler**

School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ ,U.K
{dsy04r,mjb}@ecs.soton.ac.uk

Abstract. System availability is improved by replication of data objects in a distributed database system. It is advantageous to replicate data objects when transaction workload is predominantly read only. However, during updates, the complexity of keeping replica identical arises due to failures. A number of approaches has been proposed to make systems fault tolerant through exchange of messages. Logical clocks provide the framework to realize global causal ordering on messages. The algorithms ensuring globally ordered delivery of messages may be coupled with the provisions to provide fault tolerance in event of failures. The B Method provides state based formal notations for writing specification of software systems. Event B provides a formal approach to development of such complex system. In this paper we present a part of ongoing work in this area. The specification for global ordering of messages is presented as B Machine. The global ordering of messages may be achieved by implementing Vector Clocks. The same approach may be extended to the formal development of a fault tolerant distributed data replication system.

1 Introduction

A distributed system is a collection of autonomous computer system that cooperate with each other for successful completion of a distributed computation. A distributed computation may require access to resources located at participating sites. A typical database transaction contains a sequence of database operations. This sequence of database operations is considered as an atomic action. In order to maintain the consistency of the database either all of the operations need be done or none at all. A distributed transaction may spawn several sites reading or updating data objects. The purpose of replication of data objects at different sites is to increase the availability of systems which in turn speeds up query processing. Replication of data is advantageous when the transaction work load is

* Divakar Yadav is a Commonwealth Scholar supported by the Commonwealth Scholarship Commission in the United Kingdom.

** Michael Butler's contribution is part of the IST project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

predominantly read only. However in the case of updates, it is necessary to keep the replicas identical, failing to do that may lead the database in to an inconsistent state. A distributed transaction consists of communicating transaction components at participating sites. A commit or abort decision of a distributed transaction is based on the decision of components of the transaction running at participating sites. A commit is an unconditional guarantee that update to a database are permanent. To maintain consistency of a database it is necessary that a transaction commit at each participating site or at none of sites.

There exist several approaches to ensure global atomicity. Gray addressed the issue of ensuring global atomicity despite failures in [10]. The two phase commit protocol provides fault tolerance to distributed transactions despite failures. Transaction failure, site failure and network partition are causes of failures. An increasing number of components in a distributed system imply a higher probability of component failure during execution of distributed transactions. The *two phase commit* protocol ensures global atomicity through exchange of messages among the participating sites and coordinating site. The drawback of this protocol is that it is blocking because in case of failure of the coordinator site, participants wait for its recovery. The variants of this protocol were proposed to improve the performance of this protocol in [14]. The *presumed commit* protocol is optimized to handle general update transactions while *presumed abort* optimizes partial read-only transactions. Levi and others presented an *optimistic two phase nonblocking commit* protocol [12] in which locks acquired on data object at a site are released when the site is ready to commit. In case of abort of distributed transactions, a compensating transaction is executed at that site to undo the updates. A *three phase commit* protocol [18] is a nonblocking commit protocol where failures are restricted to site failures only. All of these protocols assume that mechanisms such as maintaining the database log and local recovery are present locally at each site. There are a number of communication paradigm in which commit protocols are implemented. In *centralized two phase commit* protocol no messages are exchanged among participating sites and messages are exchanged only between between the coordinator site and cohorts. In the *nested two phase commit* protocol cohorts may exchange messages among themselves. A *distributed two phase commit* eliminates the second phase as the coordinator and cohorts exchange messages through broadcasting. Every site may reach the decision to abort/commit by means of vote-abort or vote-commit message.

In a distributed system neither a global common clock nor shared memory exist. In the absence of a global clock and shared memory, an up to date knowledge of a system is not known to any process. In these systems the processes communicate through exchange of messages. These messages are delivered after arbitrary time delays. This asynchronous distributed system model may span large geographical areas. A system may be designed as a fault tolerant system either by masking failures or by following a defined sequence of steps in the process of recovery after failure. The transaction updates are visible to concurrent transactions only if it successfully commits. The update caused by a failed transaction are not made visible to other concurrent transactions. This may be achieved if the system follows well defined steps on recovery from failures. The distributed two phase commit protocol requires broadcasting of messages among the sites. Global causal ordering of messages is used to achieve error recovery using vector clocks. A global ordering on messages may be defined by employing logical clocks. The algorithms ensuring globally ordered delivery of messages may be coupled with provisions of recovery from failures and fault tolerance in the event of process failures or network partitions. This also helps debugging distributed computations since they provides the mechanism to identify the order in which they occurred despite process failures or network partition. A good description of work on logical clocks and its application in solving varying problems of distributed computation may be found in [20], [9], [15]. There has been lot of work in development of fault tolerant protocols for distributed system, very few have been subjected to formal verification. It is desirable that model of distributed system be precise, reasonably compact and one expects that all the aspects of system must be considered in proofs because it leads to better design.

The B Method is proof based method for the rigorous development of systems. In this paper we outline the formal development of a system for global causal ordering of messages using vector clocks. This is a part of on-going work on the formal development of a fault tolerant distributed data replication system.

2 The B Method and Event B

Formal methods provides a systematic approach to the development of complex systems. Formal methods use mathematical notations to describe and reason about systems. B Method [1] a model oriented formal notation developed by Abrial. The B Method provides a state based formal notation based on set theory for writing abstract models of systems. A system may be defined as an abstract machine . Abstract machine contains *sets,variables,invariants, initialization* and a set of *operations* defined on variables. The *set* clause contains user defined sets that can be used in rest of machine. The variables describe the state of machine. The *invariants* are first order predicates and these invariants are to be preserved while updating the variables through the operations. The operations can have input and output parameters. Operation of machines are defined through generalized substitution. The B method allows specifications of abstract model to be written and support the stepwise refinement. At each refinement step we get more concrete specification of system. The B Method requires the discharge of proof obligations for consistency checking and refinement checking. Consistency checking involves showing that a machine preserves invariants when operations are invoked. Refinement checking involves showing that specifications at each refinement step are valid. The B Tools (Atelier B, Click'n'Prove,B-Toolkit) also provides an automatic and interactive prover. Typically the majority of proof obligations are proved by automatic prover, however some of the complex proof obligations needs to be proved interactively.

Though a significant work has been done in the area of message passing systems, logical clocks, recovery, checkpointing and fault tolerance yet application of proof based formal method to this work is rare to our knowledge. B can be used to provide formalization of protocols and algorithms of distributed system. Event B was introduced for modeling of distributed system. In Event B [2] operations are referred to as events which occurs spontaneously rather then being invoked. These events are guarded by predicates and these guards may be strengthened at each refinement steps. Applications of the B method to distributed system may be found in [6], [7], [17].

3 Global Ordering of Messages

A distributed program is composed of finite set of processes. The processes communicate with each other through exchange of messages. A class of problems relating such message passing system may be solved by defining global ordering on the messages. The messages are delivered to recipient process following their global order. The logical clocks such as Lamport Clock [11], Vector Clock [8] provides the mechanism to ensure globally ordered delivery of messages. A critical review of logical clocks can be found in [3], [16].

The execution of a process is characterized by sequences of events. These events can be either *internal events* or *message events*. An internal event represents a computation milestone achieved in a process, whereas message events signifies exchange of messages among the processes. *Message Sent* and *Message Receive* are message events respectively occurring at a process sending a message and receiving a message. The causal ordering of messages was proposed in [4]. Protocols proposed in [5], [19] use logical clocks to maintain the causal order of messages. A *happened before* relation defines the causal relationships between the events [11]. The happened before relation ($\rightarrow$) is defined as $a \rightarrow b$ where event a happened before b. The events a and b are either of following,

- a,b are internal events of a same process such that a , $b \in P_i$ and a happened before b.
- a,b are message events at different processes such that $a \in P_i$, $b \in P_j$, where a is *Message Send* event occurring at process P_i and b is *Message Receive* event occurred at P_j while sending a message m from process P_i to P_j.

Later we lift the *happened before* relation ($\rightarrow$) to define a global ordering on messages. The happened before relation is transitive i.e. if event a happened before b and b happened before c then a is said to happened before c.

$$a \rightarrow b \bigwedge b \rightarrow c \Rightarrow a \rightarrow c$$

We can further define the causally related and concurrent events using this relation. The two events a and b are causally related if either $a \rightarrow b$ or $b \rightarrow a$. Event a causally affects b if $a \rightarrow b$. The two events a and b are concurrent ($a \parallel b$) if $a \nrightarrow b$ and $b \nrightarrow a$. Therefore for any two events a and b there exist three possibilities i.e. either $a \rightarrow b$ or $b \rightarrow a$ or $a \parallel b$. The global ordering of messages deals with the notion of maintaining the same causal relationship that holds on *Message Send* and *Message Receive* relationship in their processes. In a broadcast network it is required that any recipient of a message must receive

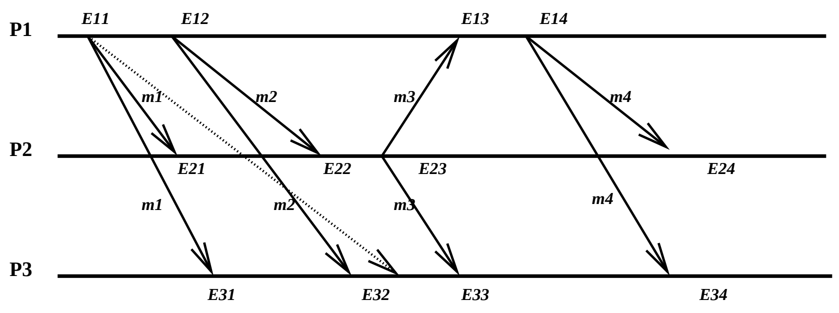


Fig. 1. Message Ordering

all the messages which are ordered before this message. As shown in figure 1, process *P1* first broadcasts message *M1*, then *P1* broadcasts the message *M2*. Process *P2* broadcasts message *M3* after receiving message *M1* and *M2* from process *P1*. Process *P1* again broadcasts message *M4* after receiving *M3* from *P2*. The global ordering among the messages may be defined on set of messages. The message *M1* is said to be ordered before *M2* as their exists causal relationship among the corresponding *Message Send* events: *Message Send* event of message *M1* happened before *Message Send* event of message *M2* in process *P1*. Therefore all recipients of message *M1* and *M2* must receive these messages in the order they were sent i.e. process *P3* must receive *M1* before receiving *M2* as shown in figure. If the message *M1* is delayed (shown as dotted lines) and delivered after delivery of message *M2*, it represents a violation of the global ordering. For any two messages M_i and M_j, message M_i is ordered before M_j ($M_i \rightarrow M_j$) if the *Message Send* event of M_i happened before the *Message Send* event of M_j and the sender of both messages is the same. If the sender process of these messages is different then message M_i will be ordered before M_j if the *Message Receive* event of M_i happened before the *Message Sent* event of M_j in the process sending message M_j. The global order of messages may be defined as follows. Suppose messages *M1* and *M2* are sent by processes *P1* and *P2* respectively. Message *M1* is ordered before *M2* (*M1* $\rightarrow$ *M2*) iff either of following holds.

– *Send(M1)* $\rightarrow$ *Send(M2)*, where sender(*M1*)=sender(*M2*) and *M1* is sent before *M2*.
– *Receive(M1)* $\rightarrow$ *Send(M2)*, where sender(*M1*) $\neq$ sender(*M2*) and *M1* is received by sender of *M2* before *M2* is sent.

Send(M) and *Receive(M)* are events representing sending and receipt of message *M* respectively. The two messages *M1* and *M2* are defined as parallel messages (*M1* $\|$ *M2*) when no partial ordering exist among them i.e. $\neg$ (*M1* $\rightarrow$ *M2*) $\wedge$ $\neg$ (*M2* $\rightarrow$ *M1*) holds. These messages may be delivered to a recipient process in any order. As shown in figure 1, *(M1* $\rightarrow$ *M2)* holds as *E11* $\rightarrow$ *E12* , *(M2* $\rightarrow$ *M3)* holds as *E22*$\rightarrow$ *E23*, *(M3* $\rightarrow$ *M4)* holds as *E13* $\rightarrow$ *E14*. Due to transitivity condition

```
MACHINE        CausalOrder
SETS           PROCESS ; MESSAGE
VARIABLES      sender, receive, order
INVARIANT
 /* Inv-1 */       sender∈ MESSAGE ⇸ PROCESS
 /* Inv-2 */     ∧ receive ∈ PROCESS ↔ MESSAGE ∧ order ∈ MESSAGE ↔ MESSAGE
 /* Inv-3 */     ∧  dom(order) ⊆ dom(sender)  ∧   ran(order) ⊆ dom(sender)
                 ∧ ran(receive) ⊆ dom(sender)
 /* Inv-4 */     ∧  ∀p,m•(p∈PROCESS ∧ m∈ MESSAGE ∧ (p↦m) ∈ receive  ⇒ p ≠ sender(m))
 /* Inv-5 */     ∧ ∀m1,m2,m3•(m1 ∈MESSAGE ∧ m2 ∈MESSAGE ∧ m3 ∈ MESSAGE
                       ∧ (m1 ↦ m2) ∈ order ∧ (m2 ↦m3) ∈ order  ⇒ (m1 ↦ m3) ∈ order)
 /* Inv-6 */     ∧∀m1,m2,p•(m1 ∈ MESSAGE ∧ m2 ∈ MESSAGE ∧ p∈ PROCESS
                       ∧ (m1↦m2)∈order ∧ (p↦m2)∈receive ∧ p≠sender(m1)  ⇒ (p ↦m1) ∈ receive )

INITIALISATION
                       sender := ∅   || receive := ∅ || order :=∅
OPERATIONS
  Send(pp,mm) ≙  PRE  pp ∈ PROCESS ∧ mm ∈ MESSAGE
                 THEN
                       SELECT mm ∉ dom(sender)
                       THEN
                             order := order  ∪( (sender~[{pp}] * {mm})  ∪ ( receive[{pp}] * {mm}))
                           || sender := sender ∪ {mm ↦ pp}
                        END
                      END;
 Receive(pp,mm) ≙ PRE  pp ∈ PROCESS ∧ mm ∈ MESSAGE
                   THEN
                       SELECT mm ∈ dom(sender)  ∧ (pp ↦ mm) ∉ receive
                                  ∧ pp ≠sender(mm)
                                  ∧ ∀m.( m ∈ MESSAGE ∧ (m↦mm) ∈ order
                                                             ∧ pp ≠ sender(m)  ⇒ (pp ↦ m) ∈ receive)
                        THEN
                                   receive := receive ∪ {pp ↦ mm}
                        END
                    END
END
```

Fig. 2. Abstract Model of Causal Order in B

(M1 → M3) ,*(M1 → M4)* and (*M2 → M4)* also holds. The abstract model of causal order of messages is presented in figure 2 as a B Model. Knowledge of B syntax is assumed. The brief description of this machine is given below.

- PROCESS and MESSAGE are defined as sets. The *sender* is a partial function from MESSAGE to PROCESS. The *receive* is a relation between PROCESS and MESSAGE ((p↦m) ∈ receive indicates that process *p* has received message *m*). The *order* is a relation between MESSAGE and MESSAGE. (Shown as *Inv-1* and *Inv-2* in the invariant clause of the model)
- A *sent* message is not received by its sender and all received message must be messages whose *Message Send* event is recorded. Similarly, ordering of messages can be defined only on those messages whose *Message Send* event is recorded. (Shown as *Inv-3,Inv-4*)
- The invariant contains a predicate which requires that transitivity property on messages should be maintained.(Shown as *Inv-5*)

- For any message whose *Message Receive* event happened at a process, that process must have received all the messages *ordered before* that message. (Shown as *Inv-6*)
- *Send* and *Receive* are events of messages defined as operations. These events are guarded by predicates. In the event of sending a message *mm* by process *pp*, all messages sent and received by process *pp* are *ordered before* the message *mm*.
- In the event of receipt of a message *mm* by a process *pp*, it must ensured that all messages *ordered before mm* has been received by process *pp*. This condition is satisfied by a predicate in the guard of operation *Receive(pp,mm)*.

4 Vector Clocks

Logical clocks are viable solution to causally order various events and to ensure globally ordered delivery of messages to processes [5], [19]. Scalar and Vector Clocks are widely referred to as logical clocks. Scalar clocks, introduced by Lamport in [11], uses an integer value to timestamp an event whereas vector clocks, introduced in [8], [13], uses a vector of integers to timestamp an event. A vector clock may be defined as a function which assign a vector of integer to an event called timestamp. For every process P_i, there exist a clock VT_{Pi} which maps an event to a vector of integer. Suppose set E_{Pi} defines the sequence of events produced by process P_i. The clock function may be defined as $VT_{Pi} : E_{Pi} \rightarrow \mathbb{V}$, where $\mathbb{V}$ is a set of vectors. The clock VT_{Pi} assigns a time stamp VT_{Pi} (e_{ij}) to event e_{ij} where $e_{ij} \in E_{Pi}$.

In a system of vector clocks, every process maintains a vector of size *N* where *N* is the total number of processes in the system. Process P_i maintains a vector clock VT_{Pi} where VT_{Pi}(i) is the local logical time at P_i while VT_{Pi}(j) represents the process P_i's latest knowledge of the time at process P_j. More precisely VT_{Pi}(j) (i$\neq$j) represents the time of occurrence of an event at process P_j when the most recent message was sent from P_j to P_i directly or indirectly. In this model, vector clocks are used to timestamp *message send* and *message receive* events only. The following rules are used to update the vector clock of process and timestamping a message in the event of *message sent* and *message receive.*

- While sending a message *M* from process P_i to P_j, sender process P_i updates its own time(i^{th} entry of vector) by updating VT_{Pi}(i) as VT_{Pi}(i) := VT_{Pi}(i) + 1. The message time stamp VT_M of message *M* is generated as VT_M(k) := VT_{Pi}(k), $\forall$ k $\in$ (1..N), where N is number of processes in system. Since a process P_i increments its own value only at the time of sending a message, VT_{Pi}(i) indicates number of messages sent out by process P_i.
- The recipient process P_j delays the delivery of message *M* until following conditions are satisfied.
  - VT_{Pj}(i)= VT_M(i) - 1
  - VT_{Pj}(k)$\geq$ VT_M(k), $\forall$k $\in$ (1..N) $\wedge$ (k $\neq$ i).

  The first condition ensures that process P_j has received all but one message sent by process P_i. The second condition ensures that process P_j has received all messages received by sender P_i before sending the message *M*. These conditions ensures global ordering on messages.

– The recipient process *Pj* updates its vector clock VT_{Pj} at *message receive* event of message *M* as VT_{Pj}(k) := max (VT_{Pj}(k), VT_M (k)). Therefore in vector clock of process P_j, VT_{Pj}(i) indicates the number of messages delivered to process P_j sent by process P_i.

Part of the B refinement of the abstract model of causal order of messages through vector clocks is shown in the figure 3 and figure 4. Figure 3 contains invariants, variables and initialization clause. The operations are shown in figure 4. A brief description of refinement steps are given below.

– Vector time of a process is represented by a variable *VTP*. The timestamp of a message is represented by a variable *VTM*. *VTP* and *VTM* are defined as functions as shown in invariant *Inv-7*, and *Inv-8*. Other conditions required for vector clock implementations are shown in invariants *Inv-10*. Vector timestamp of each process is initialized with value '0'. Initialization of variable VTP and VTM is shown in the initialization clause.
– A new variable *buffer* is introduced in refinement. The *buffer* is a relation between PROCESS and MESSAGE (*Inv-9*). The messages arriving at a process are initially buffered. The buffered messages are received by a process on satisfying the conditions as defined in vector clocks.
– The *Send*, *Arrive* and *Receive* events of a message at a process are shown as operations in figure-4. At the time of sending a message *mm*, process *pp* increments its own clock value VTP(pp)(pp) by one. The VTP(pp)(pp) represents the number of messages sent by process *pp*. The modified vector timestamp of process is assigned to message *mm* giving vector timestamp of message *mm*.
– The messages may arrive at a process in any order but their *Message Receive* event occurs at that process only if it has received all but one message

```
VARIABLES       sender, receive, order, buffer, VTP, VTM
INVARIANT
/*Inv-7*/         VTP ∈ PROCESS → (PROCESS → ℕ)
/*Inv-8*/       ∧ VTM ∈ MESSAGE ⇸( PROCESS → ℕ)
/*Inv-9*/       ∧ buffer ∈ PROCESS ↔ MESSAGE   ∧ ran(buffer)  ⊆ dom (sender)
/*Inv-10*/      ∧ ∀m1,m2,p•(m1∈ MESSAGE ∧ m2∈ MESSAGE ∧ p ∈ PROCESS
                           ∧ (m1 ↦ m2) ∈ order   ⇒ VTM (m1)(p) ≤ VTM(m2)(p) )
INITIALISATION

 VTP :=  PROCESS * { PROCESS * {0}}
||VTM := ∅
|| sender :=∅ || buffer := ∅ || receive :=∅ || order := ∅
```

Fig. 3. Refinement using Vector Clocks : Invariants

```
OPERATIONS

 Send(pp,mm) ≙    SELECT mm ∉dom(sender)
                  THEN
                             LET nVTP
                             BE   nVTP = VTP(pp) ⩤ { pp ↦ VTP(pp)(pp)+1}
                             IN    VTM(mm) := nVTP   || VTP(pp) := nVTP   END
                          || sender := sender ∪ {mm ↦ pp}
                   END ;

Arrive(pp,mm) ≙    SELECT      mm ∈ dom(sender) ∧ (pp ↦ mm) ∉ buffer
                            ∧ (pp ↦ mm ) ∉ receive   ∧ pp ≠ sender(mm)
                   THEN
                            buffer := buffer ∪ {pp ↦ mm}
                   END ;

Receive(pp,mm) ≙  SELECT    (pp ↦ mm) ∈ buffer   ∧ (pp ↦ mm) ∉ receive  ∧ pp ≠ sender(mm)
                       ∧ ∀p.( p ∈ PROCESS ∧ p ≠ sender(mm)    ⇒ VTP(pp)(p) ≥ VTM(mm)(p))
                       ∧  VTP(pp)(sender(mm)) = VTM (mm)(sender(mm))-1
                   THEN
                            receive := receive ∪ {pp ↦ mm}   || buffer := buffer - {pp ↦mm}
           ||  VTP(pp) := VTP(pp) ⩤ ({q | q ∈ PROCESS∧VTP(pp)(q) < VTM(mm)(q)} ◁ VTM(mm))

                    END
END
```

Fig. 4. Refinement using Vector Clocks : Operations

from the sender of that message. Vector timestamp of recipient process and message are also compared to ensure that all messages received by sender of message before sending it, are also received at the recipient process. These conditions are included as a guard in *Receive* operation. It can be noticed that the guard involving the variable *order* in the abstract model is replaced by a guard involving comparison of vector timestamp of message and process in the refinement.[1]

– The replacement of the guard involving variable *order* in abstract model with guards involving comparison of vector timestamp in refinement generates proof obligations. These proof obligations can be discharged interactively using a B Prover.

5 Conclusions

The abstract model in figure-2 of causal order provides a clear specification of causal ordering property on messages. We are currently working on formal development of fault tolerant distributed data replication system and we are finding that the abstract model of causal ordering is much easier to work with

[1] (f ⩤ g) represents function *f* overridden by *g*. (s ◁ f) represents function *f* is domain restricted by *s*.

than the vector clock model when proving the correctness of recovery mechanism. In this paper we outlined how the abstract causal order is in turn correctly implemented by the vector clock system. Our experience shows that abstraction and refinement are valuable techniques for modeling and verification of complex systems.

References

1. J R Abrial. The B Book : Assigning Programs to Meaning, Cambridge University Press,1996.
2. J R Abrial. Extending B without changing it.(For Distributed System).Proc. of 1st Conf. on B Method,pp 169-191,1996
3. R Baldoni, M Raynal. Fundamental of Distributed Computing : A Practical tour of Vector Clock Systems. IEEE Distributed System online, Vol 3, No2 , Sept 2002.
4. K P Birman, T A Joseph. Reliable communication in the presence of failures. ACM Transaction on Computer System,pp47-76 Vol5,No.1,1987.
5. K P Birman, A Schiper, P Stephenson. Lighweight causal and atomic group multicast. ACM Transaction on Computer System,Vol9,No 3,pp 272-314, 1991.
6. M Butler. An Approach to Design of Distributed Systems with B AMN. Technical Report,Electronics and Computer Science,Univerity of Southampton,1996
7. M Butler, M Walden. Distributed System Development in B. Proc. of Ist Conf. in B Method, Nantes,pp155-168,1996
8. C Fidge. Logical Time in Distributed Computing System. Computer, Vol 24,no 8,pp. 28-33,1991.
9. H Garcia-Molina, J D Ullman , J Widem. Database System : A complete Book. Pearson Education, 2002.
10. J N Gray. Notes of Database Operating System. in Operating System : An Advance Course, Springer Verlag,New york,pp 393-481,1979.
11. L Lamport. Time,Clocks and Ordering of events in a Distributed Sytem. Communication of ACM,Vol21,No.7,pp558-564,July 78
12. E Levy, H F Korth, A Silberschatz. An optimistic commit protocol for distributed transaction management.Proceedings of ACM SIGMOD,1991
13. F Mattern. Virtual Time and Global states of Distributed Systems. Parallel and Distributed Algorithm, Elsevier Science,North Holland,pp215-226,1987.
14. C Mohan, B Lindsey, R Obermark. Transaction management in R* Distributed Database. ACM TODS,11(4):378-396,1986.
15. M T Ozsu, P Valduriez. Distribted Database Systems. Prentice Hall,1999
16. M Raynal, M Singhal. Logical Time : Capturing casuality in Distributed System. IEEE Computer 29(2) : 49-56,IEEE,Feb 1996.
17. A Rezazadeh, M Butler. Some Guidelines for formal developement of web based application in B Method. Proc. of 4th Intl. Conf. of B and Z users,Guildford,LNCS,Springer,pp 472-491,April 2005.
18. D Skeen. Non Blocking Commit Protocol. ACM SIGMOD,Intl. Conf. on Management of Data,pp133-142,1981
19. A Schiper, J Eggli, A Sandoz. A new algorithm to introduce causal ordering. Proc. Intl. Workshop on Distributed Algorithms,Springer-Verlag,NewYork,pp 219-232,1989.
20. M Singhal, N Shivratri. Advanced Concept in Operating System. Tata McGraw Hill ,Delhi, 2001

Are Practitioners Writing Contracts?

Patrice Chalin

Dept. of Computer Science and Software Engineering,
Dependable Software Research Group, Concordia University
www.cse.concordia.ca/~chalin

Abstract. As the size and complexity of software systems continue to increase, it becomes essential to have rigorously defined component interfaces. Design by Contract (DBC) is an increasingly popular method of interface specification for object-oriented systems. Many researchers are actively adding support for DBC to various languages such as Ada, Java and C#. Are these research efforts justified? Does having support for DBC mean that developers will make use of it? We present the results of a quantitative survey that measured the proportion of assertion statements used in Eiffel contracts. The survey results indicate that programmers using Eiffel (the only active language with integral support for DBC) tend to write assertions in a proportion that is higher than in other languages.

Keywords: assertions, design by contract, survey, industrial practice, Eiffel.

1 Introduction

One of the effective ways of managing the size and complexity of modern day software systems is to use a modular design methodology. An appropriate partitioning of a system into modules (e.g., libraries, classes, etc.) offers an effective means of managing complexity while providing opportunities for reuse. But when applied to large industrial applications in general and fault-tolerant systems in particular, modular design methods can only truly be effective if module interfaces are rigorously defined.

An increasingly popular approach to interface specification for object-oriented software is referred to as Design by Contract (DBC) [Meyer97]. Support for DBC is built in to the Eiffel programming language. Although Eiffel is the only active language with integrated support for DBC, researchers are currently busy adding DBC support to other languages. For example,

- Spark for Ada [Barnes03],
- Spec# for C# [Barnett+04],
- Java Modeling Language (JML) [Burdy+04], Jass [BCMW01], Jcontract [Parasoft05], ESC/Java [Flanagan+02] and ESC/Java2 [ESCJ] for Java,
- JACK for Java (JavaCard) [BRL03]

Are such research efforts justified? For example, does having built-in support for DBC mean that developers will write contracts? In an attempt to provide initial answers to these questions we have conducted a survey of the use of contracts in

Eiffel projects. More specifically, we have sought to measure the proportion of source lines of code in Eiffel program that are assertions. Assertions are the basic ingredients of contracts. Why did we choose Eiffel programs as the subjects of our survey? Because Eiffel is the only programming language with built in support for DBC, and this, since its inception two decades ago. Also, Eiffel is primarily used to develop fault-tolerant systems rather than consumer applications [Kiniry05].

In the next section we explain the relationship between assertions, DBC and behavioral interface specifications. A brief review of Eiffel is also given, thus providing the necessary background for the understanding of metrics used in the survey. The survey method and metrics are given in Section 3. Section 4 provides the survey results. We conclude in Section 5.

2 Design by Contract and Eiffel

2.1 Assertions, Design by Contract and Behavioral Interface Specifications

Design by Contract (DBC) refers to a *method* of developing object-oriented software that was defined by Bertrand Meyer [Meyer97]. The main concept that underlies DBC is the notion of a precise and formally specified agreement between a class and its clients. Such an agreement, named *contract* in DBC, is called a *behavioral interface specification* (BIS) in its most general form. Contracts, like BISs, are expressed using assertions and take the form of class invariants and method pre- and post-conditions, among others.

DBC as a programming language feature refers to a limited form of support for BISs where assertions are restricted to be expressions that are *executable*. Hence, for example, in Meyer's Eiffel programming language an assertion is merely a boolean expression (that possibly makes use of the special `old` operator[1]). Meyer clearly identifies this as an *engineering tradeoff* in the language design of Eiffel [Meyer97]—a tradeoff that we believe is an important stepping stone from the current use of (plain) assertions in industry to the longer term objective of the adoption of verifying compilers [Hoare03b]. It is understood that this engineering tradeoff imposes a limit on the expressiveness of Eiffel assertions (e.g. absence of quantifiers[2]) but, at the same time we believe that it is precisely this tradeoff that has kept them accessible to practitioners.

How are contracts currently used in practice? A principal use for contracts, other than for documentation, is in run-time assertion checking (RAC). All systems supporting DBC also support RAC. When RAC is enabled, assertions are evaluated at run-time and an exception is thrown if an assertion fails. Various degrees of checking can be enabled—e.g. from the evaluation of preconditions only, to the evaluation of all assertions. Enabling RAC during testing,

[1] "old" operators can only occur in postconditions; "`old` *e*" refers to the pre-state value of *e*.
[2] This exclusion is due not to the quantifiers per se, but rather to the possibility of allowing quantified expressions with bound variables ranging over arbitrarily large or infinite collections.

```
indexing                                                          1
                                                                  2
  description:                                                    3
                                                                  4
   "Routines that ought to be in class BOOLEAN"                   5
                                                                  6
  library: "Gobo Eiffel Kernel Library"                           7
  copyright: "Copyright (c) 2002, Berend de Boer and others"      8
  license: "Eiffel Forum License v2 (see forum.txt)"              9
  date: "$Date: 2003/02/07 12:49:18 $"                            10
  revision: "$Revision: 1.2 $"                                    11
                                                                  12
class KL_BOOLEAN_ROUTINES                                         13
                                                                  14
feature -- Access                                                 15
                                                                  16
  nxor (a_booleans: ARRAY [BOOLEAN]): BOOLEAN is                  17
      -- N-ary exclusive or                                       18
   require                                                        19
      a_booleans_not_void: a_booleans /= Void                     20
   local                                                          21
      i, nb: INTEGER                                              22
   do                                                             23
      i := a_booleans.lower                                       24
      nb := a_booleans.upper                                      25
      from until i > nb loop                                      26
         -- Lines 27 … 37 removed                                 27
      end                                                         38
   ensure                                                         39
      zero: a_booleans.count = 0 implies not Result               40
      unary: a_booleans.count = 1 implies                      >> 41
         Result = a_booleans.item (a_booleans.lower)              41
      binary: a_booleans.count = 2 implies                     >> 42
         Result = (a_booleans.item (a_booleans.lower) xor      >> 42
                    a_booleans.item (a_booleans.upper))           42
      -- more: there exists one and only one `i' in            >> 43
         a_boolean.lower..a_boolean.upper so that              >> 43
                    a_boolean.item (i) = True                     43
   end                                                            44
end                                                               45
```

Figure 1, Sample Eiffel class (kl_boolean_routines.e)

particularly integration testing, is an effective means of detecting bugs in modules and thus can help contribute to the increase in overall system quality.

Of course, for most applications, particularly fault tolerant systems, it is preferable to be able to guarantee the absence of assertion failures before a component is run. Extended Static Checking (ESC) tools can be used for this purpose. An ESC tool attempts to determine the validity of assertions by static analysis. ESC tools exist for Modula-3 and Java, and one is currently under development for Eiffel.

2.2 Eiffel: a brief review

A sample Eiffel class taken from the Gobo Eiffel kernel library is given in Figure 1 (some of the lines were too long to fit on the page and hence their content has been wrapped, and indented to aid in readability, at those points marked with `>>`). Classes optionally begin (and/or end) with an *indexing clause* that offers information about the class. In other languages this is often accomplished by using a comment block. Comments, like in Ada, start with a "`--`" and run until the end of the line. An Eiffel class generally declares a collection of *features* (attributes and methods). Our sample class declares only one feature, an *n*-ary exclusive or, `nxor`.

```
from
  initialization_instructions
invariant
  assertion
variant
  variant
until
  exit_condition
loop
  loop_instructions
end
```

Figure 2, Eiffel loop instruction

Of main concern to us in this paper are assertions. An *assertion* in Eiffel is written as a collection of one or more optionally tagged assertion clauses. The meaning of an assertion is the conjunction[3] of its assertion clauses. The tags can help readability and debugging (since they can be printed when the clause is violated) [Mitchell+02]. Tags `zero`, `unary` and `binary` adorn lines 40, 41 and 42 of Figure 1, respectively.

An *assertion clause* is either a

- boolean expression (e.g. line 40) or a
- comment (e.g. line 43).

As will be noted later we will count such comments as *informal assertion clauses*, or simply informal assertions. Boolean operators consist of the usual negation (`not`), conjunction (`and`), and disjunction (`or`). Eiffel also has conditional, i.e. short-circuited, conjunction (`and then`) and disjunction (`or else`). An implication operator *a* `implies` *b* is an abbreviation for (`not` *a*) `or else` *b*. Assertions can contain calls to methods identified as queries. A particular characteristic of queries is that they are not permitted to have side-effects [Mitchell+02].

In Eiffel, an assertion can be used to express a

- precondition (introduced by the keyword **`require`**),
- postcondition (**`ensure`**),
- class invariant (**`invariant`**),
- loop invariant (**`invariant`**),
- check (**`check`**)

A sample precondition is given in lines 19-20 of Figure 1. The sample postcondition (lines 39-43) illustrates the use of more than one assertion clause. Assertions in postconditions can contain occurrences of the special operator **`old`**. For example, the postcondition

```
ensure  count = old count + 1
```

will be true when the pre-state value of `count` is one less than the post-state value of `count`. A `check` is equivalent to an `assert` statement in other languages such a Java and C++.

There is only one looping construct in Eiffel and it has the general form given in Figure 2. As was previously mentioned, an assertion can be used to express a loop invariant. Also, of interest is the loop variant: an integer expression that

[3] Actually, clauses are jointed by a conditional conjunction named "and then" in Eiffel.

must decrease through every iteration of the loop while remaining nonnegative. That essentially covers the basics of what we need to be able to explain the metrics.

3 Survey

3.1 Projects

During the initial portion of our study we gathered metrics from free Eiffel software, consisting of both free commercial software (such as the source distributed with ISE's Eiffel compiler) as well as open source projects. This allowed us to fine-tune our metrics gathering tool and essentially conduct a pilot study before soliciting the participation of industry.

3.2 Metrics

Our basic metric is a count of Lines of Code (LOC) per class file. Each LOC is classified as either:

- blank line, containing at most white space, or
- comment line, containing a comment possibly preceded by white space, or
- (physical) Source Line of Code (SLOC) [Park92].

Roughly speaking our goal is to count the number of LOC that are assertions (AsnLOC) so as to be able to determine their proportion relative to the total SLOC.

Our overall count of AsnLOC will be computed from the total SLOC that are assertions as well as the total LOC that are informal assertions (IALOC)—i.e. assertions given in the form of comments. We count informal assertions because we believe that they are just as important as formal assertions in documenting contracts. In measuring the proportion of LOC that are assertions we will use the following formula:

$$\mathit{total}(\text{AsnLOC}) \,/\, \mathit{total}(\text{AdjSLOC})$$

where

$$\mathit{total}(\text{AdjSLOC}) \;=\; \mathit{total}(\text{SLOC}) + \mathit{total}(\text{IALOC}) - \mathit{total}(\text{IdxSLOC})$$

IdxSLOC is a SLOC that occurs in an indexing clause. We omit IdxSLOC lines because these lines merely provide documentation for the class in a manner that is handled by a comment block in other languages. We will keep separate AsnLOC counts for preconditions, postconditions, class invariants, checks clauses and loop variants and invariants. This will allow us to determine the proportion of assertions used in each of these categories. We will also collect specialized metrics such as the number of assertions of the form *e* `/= Void`. Their purpose will be explained in the next section.

Project Category	Number of files	LOC (10^6)	SLOC (10^6)	% of total SLOC
Proprietary	18584	2.65	2.03	51%
Open Source	10657	1.76	1.31	33%
Eiffel 5.5	4840	0.95	0.66	17%
Total	34081	5.37	4.00	100%

Figure 3, General metrics by project category

	LOC	SLOC	blank	comment	IdxSLOC
Total (10^6)	5.37	4.00	0.818	0.546	0.171
% LOC	100%	74.6%	15.2%	10.2%	3.18%

Figure 4, General metrics (all categories)

3.3 Methodology

Initially we used the SLOCCount tool [Wheeler05] as our base. This tool can count physical SLOC for over two dozen languages—though initially not for Eiffel. Aside from its ability to process many different kinds of languages SLOCCount also does convenient house-keeping tasks such as determining the type of a file (by its extension or content), flagging duplicates, and ignoring generated files.

Since our needs were specific to Eiffel source, we eventually chose to use a single Perl script to gather all metrics. The creation of the script did pose some challenges due, e.g., to the various flavors of Eiffel (as supported by different compilers) and inconsistent line endings (Unix, DOS or Mac) sometimes in the same file.

4 Results

Overall we surveyed 81 projects totaling 34081 Eiffel class files, 5.4 million lines of code (LOC) and 4.0 million source lines of code (SLOC). Each project we developed by a different group or organization. We divided the projects into three categories:

- proprietary,
- open source and
- library and samples shipped with ISE Eiffel Studio 5.5.

Note that half of the files in the Eiffel 5.5 category consist of open source samples (or what they call free add-ons) most of which are provided by GoboSoft—an important contributor of open source Eiffel libraries and tools. The proportion of SLOC per project category is given in Figure 3. Figure 4 provides the overall distribution of LOC into SLOC, blank lines and comments. The IdxSLOC is the proportion of SLOC that occur in indexing blocks.

Metrics concerning assertions are given in Figure 5. Overall, there were 89468 lines of assertions (AsnLOC) out of 3.84 million SLOC (AdjSLOC); that is, 4.39% of the LOC are assertions (AsnLOC/AdjSLOC). Of this, over 50% are

	require	ensure	inv. class	inv. loop	var. loop	check	Total
AsnLOC	89468	61267	10882	332	325	6206	168480
AsnLOC/AdjSLOC	2.33%	1.60%	0.28%	0.01%	0.01%	0.16%	4.39%
LOC/AsnLOC	53.10%	36.36%	6.46%	0.20%	0.19%	3.68%	100.00%
IALOC	1129	3710	996	91	0	502	6428
IALOC/AdjSLOC	0.03%	0.09%	0.02%	0.00%	0.00%	0.01%	0.16%
IALOC/AsnLOC	0.67%	2.20%	0.59%	0.05%	0.00%	0.30%	3.82%
No. of clauses	53677	39550	4614	203	324	5139	103507
Count(*e* /= Void)	39742	14484	5571	7	0	2177	61981
							Average or max
Max AsnLOC size	30	24	35	7	2	14	35 (max)
Average size	1.7	1.5	2.4	1.6	1.0	1.2	1.6
% (*e* /= Void)	44.42%	23.64%	51.19%	2.11%	0.00%	35.08%	36.79%

Figure 5, Metrics concerning assertions (all project categories)

Project Category	SLOC (10^6)	AdjSLOC (10^6)	AsnLOC (10^6)	AsnLOC / AdjSLOC
Proprietary	2.03	1.96	0.064	3.27%
Open Source	1.31	1.25	0.064	5.10%
Eiffel 5.5	0.66	0.63	0.040	6.42%
Total	4.00	3.84	0.168	4.39%

Figure 6, Proportion assertion LOCs per project category

used in preconditions, 36% postconditions, and 6.5% class invariants. Few loop invariants and variants are given, though both of these appear as frequently, relative to each other. We note that a very small proportion of assertions are given in the form of comments; i.e. overall, only 3.8% of assertion LOC are informal assertions (IALOC). The maximum number of assertions per clause type can be fairly large—e.g. up to 35 LOC for a class invariant. The average number of assertions per clause type ranges from 1.0 to 2.4.

A noteworthy proportion of assertions include subexpressions of the form *e* `/= Void` asserting that a given reference is not `Void` (i.e. null). This number is over 50% for class invariants, but 37% overall. Such figures may provide weight to the choice by some static analysis tools (such as Splint [Evans03]) to assume that a reference type declaration is non-null by default[4].

Finally, in Figure 6 we show how the proportion of LOC that are assertions (AsnLOC) varies according to the project category. As might be expected, the Eiffel category has the highest proportion, 6.4%, followed by open source projects[5] and then proprietary code with a little over 5% and 3%, respectively.

[4] Our research group is currently examining the possibility of adopting such a default in the Java Modeling Language (JML). In JML non-null declarations appear even more frequently.
[5] Recall that the open source category *excludes* GoboSoft software (since it is counted in the Eiffel 5.5 project category).

5 Conclusion

We concede that the Eiffel survey sampling is not very large by industrial standards (5.4 MLOC), but we are hopeful that it is somewhat representative. Our survey is still ongoing—both for Eiffel and other languages. On the other hand, we do anticipate that the use of Eiffel will be significantly less than the use of, e.g. C or C++. The relatively small size of the Eiffel user community may also have some bearing on the survey results—e.g. a lesser variability.

Our survey data focuses on the use of assertions in Eiffel, the only active language supporting the disciplined use of assertions in specifying contracts, i.e. Design by Contract (DBC). Before conducting the survey we asked: does having language support for DBC mean that practitioners will make use of it? Overall, 4.4% of the (physical) SLOC of the surveyed projects were assertions. The results for the category of projects consisting solely of proprietary code was 3.3%. This is almost twice as much, for example, as the percentage of assertions reportedly used in the Microsoft Office Suite [Hoare00, Hoare03a] as well in a separate independent study we have conducted [Chalin05]. In our opinion, this is good news for those researchers currently striving to add DBC support to other languages.

By design, DBC restricts the expressiveness of assertions by requiring that they be executable. We believe that this moderation in expressiveness is what will allow DBC to be more easily adopted by industry. It will then become a smaller step to reach the full expressiveness of behavioral interface specifications (BISs). Of course, BISs are not the entire picture either; future generation verification compilers are likely to include support for model checking as well as BISs.

References

[Barnes03] John Barnes. High Integrity Software: The Spark Approach to Safety and Security. Addison-Wesley, 2003.

[Barnett+04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. CASSIS 2004 post-proceedings.

[BCMW01] D.Bartetzko, F. Clemens, M. Möller, and H. Wehrheim. "Jass – Java with Assertions." *Electronic Notes in Theoretical Computer Science* 55(2), 2001.

[Bicarregui98] J. Bicarregui, editor. *Proof in VDM: Case Studies*, Springer-Verlag, FACIT series, March 1998.

[Burdy+04] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino and Erik Poll. An overview of JML tools and applications. In *International Journal on Software Tools for Technology Transfer* (STTT), 2004.

[Chalin05] Patrice Chalin *Logical Foundations of Program Assertions: What do Practitioners Want?* Proceedings of the Software Engineering and Formal Method Conference 2005. Koblenz, Germany September 5-9, 2005, to appear.

[Evans03] David Evans. *Splint User Manual*. Secure Programming Group, University of Virginia. June 5, 2003. www.splint.org.

[Flanagan+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI-02)*, ACM SIGPLAN 37(5):234–245, June 2002.

[Hoare00] C.A.R. Hoare, *Assertions Progress and prospects*. Presentation available from research.microsoft.com/~thoare.

[Hoare03a] C.A.R. Hoare, Assertions: a personal perspective. *Annals of the History of Computing*, IEEE 25(2):14-25, April-June 2003.

[Hoare03b] C.A.R. Hoare, The Verifying Compiler: A Grand Challenge for Computing Research. *JACM*, 50(1):63-69, 2003.

[Kiniry05] J. Kiniry, Chair of NICE (Non-profit International Consortium for Eiffel). Personal communication, June 2005.

[Kramer98] Reto Kramer. iContract—the Java Designs by Contract tool. In *Proc. Technology of Object-Oriented Languages and Systems, TOOLS 26*. IEEE Press, 1998.

[Meyer97] Bertrand Meyer. *Object-Oriented Software Construction*. 2nd edition. Prentice Hall, 1997.

[Mitchell+02] Richard Mitchell, Jim McKim. *Design by Contract, by Example*. Addison-Wesley, 2002.

[Park92] R. Park, “Software Size Measurement: A Framework for Counting Source Statements.” CMU/SEI-92-TR-20, Software Engineering Institute, Pittsburgh, PA, 1992.

[Parasoft05] Jcontract product page available at www.parasoft.com.

[Wheeler04] David A. Wheeler, www.dwheeler.com/sloccount.

Modeling and Analysis of Architectural Exceptions

Fernando Castor Filho[⋆], Patrick Henrique da S. Brito[⋆⋆], and Cecília Mary F. Rubira[⋆⋆⋆]

Institute of Computing - State University of Campinas
P.O. Box 6176. CEP 13083-970, Campinas, SP, Brazil.
{fernando, patrick.silva, cmrubira}@ic.unicamp.br
+55 (19) 3788-5842 (phone/fax)

Abstract. In recent years, many approaches combining software architectures and exception handling have been proposed for increasing the dependability of software systems. Some authors argue that addressing exception handling-related issues since early phases of a software development effort may improve the overall dependability of a system. In particular, few works in the literature have addressed the problem of describing how exceptions flow between architectural components. This is an important issue, since developers tend to focus on the design of the normal activity of the system'components and address its exceptional activities only during the implementation phase. A model for describing the flow of exceptions between architectural components should be: (i) precise; and (ii) analyzable, preferably automatically. In this paper, we present a model for reasoning about exception flow in software architectures that satisfies these two requirements. The model is supported by a software infrastructure which leverages existing tools and models and allows developers to describe and analyze software architectures enriched with information about exceptions and their flow.

1 Introduction

The concept of software architecture [7] was proposed in the last decade to help software developers to cope with the growing complexity of software systems. According to Clements and Northrop [7], software architecture is the structure of the components of a program/system, their interrelationships and principles and guidelines governing their design and evolution over time. The architecture of a software system has a large impact on the capacity of the system to meet its intended quality requirements, such as reliability, security, availability, and performance, among others. Software architectures are described formally using architecture description languages, or ADLs [18]. ADLs share the same conceptual basis whose main elements are components (loci of computation or data stores), connectors (loci of interaction between components), and configurations (connected graphs of components and connectors that describe architectural structure) [18].

[⋆] Supported by FAPESP/Brazil under grant 02/13996-2.
[⋆⋆] Supported by the Specialization in Software Engineering course, IC/UNICAMP.
[⋆⋆⋆] Partially supported by CNPq/Brazil under grant 351592/97-0.

When a program[1] receives a service request and produces a response according to its specification, the produced response is said to be *normal*. Conversely, if the program produces a response that does not conform with its specification, this response is said to be *abnormal*, or *exceptional*. Abnormal responses usually indicate the occurrence of an error and since these responses are expected to occur only rarely, they are called *exceptions*. When exceptions occur, the program must be capable of handling them so that it can be put in a coherent state. The part of the behavior of a program that is responsible for handling exceptions is called *abnormal*, or exceptional, *activity*. Conversely, the part of the behavior of a program that is responsible for its functionality, as defined by its specification, is called *normal activity*.

Exception handling [8] is a mechanism for structuring the exceptional activity of a program, so that errors can be more easily detected, signalled, and handled. Since exception handling is an application-specific technique, it complements other techniques for improving system reliability, such as atomic transactions, and promotes the implementation of very specialized and sophisticated error recovery measures.

Problem Description. In recent years, many approaches combining software architectures and exception handling [4][13][20] have been proposed for increasing the dependability of software systems. We say that an exception is *architectural* if it is raised within an architectural component but can not be handled by the raising component. Such exceptions cross the boundaries between architectural components, that is, the architectural exceptions that flow between two components are part of the interaction protocol to which the two components adhere. Combining software architectures and exception handling is a natural trend. The architecture of a software system has a large impact on a system's quality attributes, such as reliability, and architectural exceptions indicate that architectural components have failed (and have thus been unreliable).

There are many works proposing notations and techniques for describing software architectures formally [1][11][17] focusing on specific properties of interest. However, to the best of our knowledge, few have addressed the problem of describing how architectural exceptions flow between components. As pointed out by Bass et al [2], specifying how exceptions flow between architectural components is a real problem that appears in the development of systems with strict dependability requirements, such as air-traffic control and financial. To be useful and usable, an approach for describing architectural exceptions and their relationship to other architectural elements must satisfy some requirements:

1. It should make it possible to specify the architectural exceptions that components and connectors signal and catch, and how these exceptions flow between different architectural elements. Ideally, the specification of the architectural exceptions of the system should be orthogonal and traceable to the "normal" architecture description, in order to enhance maintainability.
2. It should have pictorial (boxes-and-lines) representation, in order to be understandable by non-specialists and easier to use.

[1] In a general sense: a routine, a software component, a whole system, etc.

3. It should take into account the notion of *architectural styles*. An architectural style defines a vocabulary of types of design elements which are part of a family of architectures and the rules by which these elements are composed [11].
4. It should be precise, that is, an architecture description devised according to such approach should be unambiguous.
5. It should be expressive enough to describe rules of existing exception handling models.
6. It should be analyzable, preferably automatically. In this manner, it is possible to verify if the architecture presents some desired properties before the system is actually implemented.

Proposed Approach. In another work [5] we have proposed a framework, called Aereal, that addresses these requirements. In that work, we focused on requirements 1, 2, and 3. In this work, we present a model for reasoning about exception flow in software architectures that addresses requirements 4, 5, and 6. The proposed model is part of the Aereal framework. It allows developers to specify common rules of exception handling systems (EHS) of existing programming languages and to verify in an automated way if an architecture description extended with information about architectural exceptions adheres to these rules. As enabling technology, we use the Alloy [14] specification language and the Alloy Analyzer [15].

This work is organized as follows. Section 2 presents the proposed model in terms of three aspects: system structure, representation of exceptions, and exception flow. Section 3 briefly describes how we have materialized the proposed model using Alloy in the Aereal framework. The last section compares the proposed model with some related research and presents directions for future works.

2 Proposed Model

The set of exceptions and exception handlers in a program define its exceptional activity. When an error is detected, an exception is generated, or *raised*. If the same exception may be raised in different parts of a program, it is possible that different handlers are executed. The choice of the handler that is executed depends on the exception handling context (EHC), or scope, where the exception was raised. An EHC is a region of a program where the same exceptions are handled in the same manner. Each context has an associated set of handlers that are executed when the corresponding exceptions are raised. An exception raised within an EHC may be caught by one of its handlers. If the exception is *handled*, normal activity of the program is resumed. Otherwise, an exception is *signaled* in the enclosing context, and *encountered* by that context. We assume that EHCs only encounter a single exception at a time. Concurrent exceptions [3] are not addressed by this work.

In this section, we present the proposed model using a mix of informal explanations, and set theory notation. Due to space constraints, we omit some parts of the description of the model. A more detailed presentation is available elsewhere [6]

2.1 System Structure

We follow the general view of a system configuration as a finite connected graph of components and connectors [18]. We specialize this view, however, so that it can be used to reason about exception flow. In our model, a component is a structural element that encounters and signals exceptions.

Aereal uses special-purpose architectural connectors to model exception flow between components. These connectors, called exception ducts, are unidirectional point-to-point links through which only exceptions flow. They are orthogonal to "normal" architectural connectors and do not constrain the way in which the architecture is organized [5]. Exception ducts can be refined by developers, depending on the restrictions each architectural style imposes on exception flow. Like components, exception ducts can signal and encounter exceptions.

The structure of a system is defined in terms of connections between components and exception ducts. The relations $CatchesFrom$ and $SignalsTo$ specify these connections. For a component C, $C.CatchesFrom$ yields the set of exception ducts that signal exceptions that C encounters, where "." represents relational join. Conversely, $C.SignalsTo$ yields the set of exception ducts that encounter exceptions that C signals. Both $C.SignalsTo$ and $C.CatchesFrom$ may yield an empty set, in which case C does not signal and does not encounter exceptions, respectively.

The relations $SignasTo$ and $CatchesFrom$ are also defined for exception ducts[2]. However, since exception ducts are point-to-point connectors that link exactly two distinct components, for a duct D, $D.CatchesFrom$ and $D.Signals$ result in disjunct sets containing exactly one component. Therefore, for any component C, $D.CatchesFrom = \{C\} \Rightarrow D \in C.SignalsTo$, and $D.SignalsTo = \{C\} \Rightarrow D \in C.CatchesFrom$.

2.2 Representation of Exceptions

In our model, exceptions are represented by objects of a certain type. We represent exceptions as objects, instead of using symbols or global variables, mainly because objects are more flexible and can be used to encode arbitrary information regarding the cause of an exception [10]. Moreover, many large and complex software systems are developed nowadays using object-oriented (OO) languages such as Java, C#, and C++.

The proposed model employs a simple notion of type that is compatible with the general notion of types adopted by modern OO languages. A type T is a set of elements and the subtypes $T_1, T_2 ... T_N$ of T are disjunct subsets of T. Only single inheritance is allowed. An exception is any instance of a type that is a subtype of the type RootException. We use this name for the supertype of all exceptions, instead of a more usual one, such as Exception or Error, to give developers the flexibility to organize exceptions as required, for instance, based on the adopted programming language. For example, to mimic the EHS of Java, a developer would define at least four exception types: (i) Throwable, subtype of RootException; (ii) Exception, subtype of Throwable; (iii) Error, subtype of Throwable; and (iv) RuntimeException, subtype of Exception.

[2] Actually, we use overloaded relation names as a syntactic sugar, since the homonymous relations have very similar semantics.

2.3 Exception Interfaces and Exception Handling Contexts

As mentioned in previously, we consider a component to be a structural element that encounters and signals exceptions. Exception ducts are similar, but simpler. A component consists of: (i) a collection of exception interfaces, which specify the exceptions the component signals; and (ii) a collection of EHCs, which define regions where exceptions are always handled in the same way. In this section and the next, for space reasons, we focus our attention exclusively on the definition of an exception flow model for components. Exception ducts are described in a more complete version of this paper, available as a technical report [6].

Exception interfaces are associated to components by the $SignalsTo$ relation and, for each exception duct in the set $C.SignalsTo$, there is a corresponding exception interface. A similar one-to-one relation exists between $CatchesFrom$ and EHCs. This represents the fact that a component may signal/encounter different exceptions to/from the different exception ducts it is connected to. Models for reasoning about exception flow at the programming language level usually do not have this separation between interfaces and contexts. Such separation is not necessary because these models usually focus on fine-grained programming constructs, like methods and procedures, where multiple contexts are associated to a single exception interface. For architectural exceptions, however, this separation is very important, since a component can have multiple access points (ports) and these access points are explicit in the system description.

In our model, exception interfaces and EHCs are related by the $PortMap$ relation. $PortMap$ maps EHCs to exception interfaces based on the exception ducts to which these contexts and interfaces are associated. For any component C and exception duct D, with $D \in C.CatchesFrom$, $D.(C.PortMap) = DS$, where DS is a set of exception ducts such that $\forall X : DS \bullet X \in C.SignalsTo$. DS is a set of exception ducts, instead of a single duct, to represent the fact that the association between EHCs and exception interfaces is many-to-many. It makes no sense to define a $PortMap$ relation for exception ducts, since they have exactly one EHC and one exception interface.

2.4 Exception Flow

The exception interfaces of a component are defined by the $Signals$ relation. This relation specifies which exceptions a component signals and which exception ducts in $C.SignalsTo$ encounter these exceptions. If $D.(C.Signals) = ES$, where ES is a set of exceptions, we say that component C **signals** exceptions ES to duct D. The $signals$ relation is defined in terms of three other relations, as follows:

$$Signals \ = \ Raises \bigcup Propagated \bigcup Unhandled$$

Intuitively, the set of exceptions that a component signals depends on the exceptions it generates (raises) and on exceptions it encounters that were signaled by other architectural elements. The $Propagated$ and $Unhandled$ relations are auxiliary relations defined in terms of the relations that specify a component's EHCs (described in the following paragraphs). The $Raises$ relation specifies the exceptions that components generate when erroneous conditions are detected. These conditions are dependent on the semantics of the application and on the assumed failure model. For reasoning about exception flow, the fault that caused an exception to be raised is not important, just the fact that the exception was raised. If $D.(C.Raises) = ES$, we say that the component C **raises** exceptions ES and these exceptions are signaled to exception duct D, where $D \in C.SignalsTo$ and $ES \subset D.(C.Signals)$. More generally, $Raises \subset Signals$.

Exception handling contexts are defined in terms of three relations: $Encounters$, $Handles$, and $Propagates$. $Encounters$ specifies, for an arbitrary component C, the exceptions C receives from the exception ducts in the set $C.CatchesFrom$. That is, if $D.(C.Encounters) = ES$, we say that the component C **encounters** exceptions ES that were signaled by exception duct D. In fact, the set of all exceptions encountered by component C is equal to the union of the sets of exceptions signaled to C by exception ducts in $C.CatchesFrom$. In this sense, the definition of $Encounters$ used'in our model is different from the definitions adopted in other works in the literature [19, 22].

The $Handles$ relation specifies the exceptions that are handled by a component. By "handled", we mean that the component is capable of taking some action that stops the propagation of the exception and makes it possible for the system to resume its normal activity. The action that is taken by the handler is not important in the context of this work. We are just interested in the effect the handler has on the flow of exceptions, not how this effect is achieved[3]. If $D.(C.Handles) = ES$, we say that the component C **handles** the exceptions ES signaled by exception duct D, where $D \in C.CatchesFrom$ and $ES \subset D.(C.Encounters)$. More generally, $Handles \subset Encounters$.

In our model, the $Propagates$ relation explicitly specifies a causal relation between an exception a component encounters and another one it signals. More precisely, if $E.(D.(C.Propagates)) = E$', where E and E' are exceptions, we say that component C **propagates** exception E', signaled by exception duct D as E, with $D \in C.CatchesFrom$, $E \in D.(C.Encounters)$, $E \notin D.(C.Handles)$, and E'$\in (D.(C.PortMap)).(C.Signals)$. The latter constraint states that C signals the propagated exception (E') to the exception ducts related to D in $C.PortMap$. If $E.(D.(C.Propagates)) = \{\}$ and $E \notin D.(C.Handles)$, it is assumed that $E.(D.(C.Propagates)) = E$ and E is signaled to all exception ducts D' such that D'$\in C.SignalsTo \wedge D$'$\in D.(C.PortMap)$.

Now we can go back to the definition of $Signals$ and define $Propagated$ and $Unhandled$. $Propagated$ specifies the exceptions that a component signals due to exception propagation (unlike $Propagates$, which relates two exceptions, one encountered and one signaled by the component). For a component C and an exception duct D and using DS as a shortcut for $(C.PortMap).D)$, $Propagated$ is defined by the following expression:

$$D.(C.Propagated) = (DS.(C.Encounters \setminus C.Handles)).(DS.(C.Propagates))$$

$Unhandled$ specifies the set of exceptions that a component encounters but does not propagate explicitly (as specified by $Propagated$) or handle. Like in some programming languages, such as Java, the exceptions which are not either handled nor ex-

[3] We are not stating that the way an exception is handled is not important. Just that modeling the actual exception handlers is beyond the scope of this work.

plicitly propagated ($Propagated$) are automaticaly propagated. These exceptions are signaled by the component (propragated implicitly). For a component C and a duct D and using the same shortcut defined in the previous paragraph, $Unhandled$ is defined as follows:

$$D.(C.Unhandled) \; = \; (DS.(C.Encounters \setminus C.Handles)) \setminus \\ ((DS.(C.Propagates)).(D.(C.Propagated)))$$

The last element of our model is the $Declares$ relation. $Declares$ is used to make the exception interfaces of components in a system explicit. This relation is part of the model to allow developers to explicitly state which exceptions each component signals, for instance, because the programming language that will be used to implement the system has a similar feature. If $D.(C.Declares) = E$, we say that component C **declares** that exception E is signaled to exception duct D or simply E is one of the declared exceptions of component C. Explicit exception interfaces are part of several modern programming languages, for example, Java, C++, and ML. By default, $Declares$ does not impose any constraints on exception flow. An analysis can, however, take the relation into account in order to impose some application- or EHS-specific constraints.

3 Materializing the Model

Usually, models like the one described in this paper are used as the backbone for static analysis tools [19][22]. These tools are capable of extracting useful exception flow-related information from programs and showing that these programs present some properties of interest, for example, that exceptions are not caught by subsumption[4]. In this work, instead of building a new tool with a fixed set of functionalities, we translated the semantic description presented in Section 2 to Alloy [14]. Alloy is a lightweight modeling language for software design. It is amenable to a fully automatic analysis, using the Alloy Analyzer (AA) [15], and provides a visualizer for making sense of solutions and counterexamples it finds. The analysis performed by the AA is sound, since it never returns false positives, but incomplete, since the AA only checks things up to a certain scope. However, it is complete up to scope; AA never misses a counterexample which is smaller than the specified scope.

In the proposed approach, systems are modeled by specifying exception types, components and exception ducts (including the relations described in the previous section), and connections between these architectural elements. The following snippet shows part of a trivial model with two components, one exception duct, and one exception type.

```
sig E extends RootException{}
sig C1,C2 extends Component{} //components extend "Component"
sig D1 extends Duct{} //exception ducts extend "Duct"
fact SystemStructure{
  C1.SignalsTo = D1
  ...
```

[4] An exception `E` is caught by subsumption if it is caught by a `catch` clause that targets a supertype `E'` of `E`.

```
}
fact ExceptionFlow{
  C1.Raises = D1 -> E
  ...
}
fact PortMaps{ (...) }
```

In Alloy, a signature (`sig` keyword) specifies a type. We use signatures for modeling both structural elements and exceptions. The relations defined in Section 2 are explicitly instantiated by means of facts, predicates that the AA must assume to be true when evaluating constraints. For instance, the fact `SystemStructure` in the snippet above states that component `C1` signals exceptions to exception duct `D1`. Moreover, the fact `ExceptionFlow` states that the component `C1` raises exception `E` to exception duct `D1`.

Using Alloy to materialize our model makes it possible to specify features of diverse EHS without having to extend existing tools or build new ones when a new feature is required. Developers need only to specify new Alloy constraints or modify the existing ones and the AA can be used to check if such constraints hold. Until the present moment, we have successfully specified several features available in existing EHS and static analysis tools, including: (i) explicit exception propagation; (ii) detection of exception subsumption; (iii) checked and unchecked exceptions; and (iv) checked exception interfaces. Section 3.1 presents a simple example.

In order for the proposed model to be of practical use, it must be integrated with some notation for describing software architectures. In this work, we have chosen the ACME [11] ADL as the notation for describing software architectures. The Aereal framework leverages ACME and its tool support [23] in order to allow developers to extend architecture descriptions with information about architectural exceptions. Aereal includes a model-to-model transformation tool that generates Alloy models from these extended architecture descriptions. The generated models can be provided "as-is" to the AA in order for analyses to be performed. Since we have already specified many of these analyses, in general developers do not need to know Alloy to use the framework.

3.1 An Example: Explicit Exception Propagation

In this section we show how a rule adopted by several EHS [12, 16], explicit exception propagation, can be described using the proposed model. At the programming language level, when a component encounters an exception, if it has a handler for the exception, this handler may re-raise the exception or raise a new one. If the component does not have a handler, there are two possible outcomes, depending on the programming language: (i) the exception is implicitly re-raised by the underlying runtime system; or (ii) an error occurs either at compile time [12] or run time [16]. In the former case, exception propagation is said to be *implicit*. In the latter, it is said to be *explicit*. Some languages, like Java, are actually hybrid and allow both implicit and explicit propagation of exceptions. For simplicity, we do not take hybrid approaches into account.

Informally, we can specify explicit exception propagation as follows: for any component `comp` in a given model, any exception encountered by `comp` and not handled should be explicitly propagated. The following Alloy predicate formally specifies explicit propagation, according to the proposed exception flow model:

```
pred explicit_propagation_component() {
 all C: Component | let nonHandled = (C.Encounters - C.Handles)
  | (all CF : C.CatchesFrom | #(CF <: nonHandled) > 0 =>
     ((#nonHandled > 0 => #(C.Propagates) > 0) &&
      all e: CF.nonHandled |  #(e.(CF.(C.Propagates))) > 0))
}
```

The snippet above defines an Alloy predicate called `explicit_propagation`. Alloy predicates are logic sentences that must be checked by the AA. When predicates are checked, Alloy facts are used as preconditions. The names of relations that appear in the example above refer to the relations described in Section 2. For example, $Encounters$ is a ternary relation of the form $Component \iff Duct \iff Exception$, indicating that a component encounters a certain exception signaled by a certain exception duct. The predicate associates a local variable, `nonHandled`, to the set of pairs of the form `(CF, E)`, where `CF` is an exception duct and `E` is an exception, such that `E` is not handled by component `C`. It then states that, for all such pairs, the expression `E.(CF.(C.Propagates))` yields some element. The interested reader is referred to the Alloy Tutorial [15] in order to understand the details of the predicate.

Since the exception flow model has been translated to Alloy, it becomes possible to check rules such as the one above automatically, using the AA. If the model does not satisfy the specified rule, the AA produces a counterexample showing why that happened.

4 Related and Future Work

The works of Fähndrich et al [9], Robillard and Murphy [19], and Schaefer and Bundy [22] describe static analyses for computing the $encounters$ relation in programs written in ML, Java, and Ada, respectively. Our work focuses on defining a model that is flexible enough for defining characteristics of real EHS used in different languages, assuming that $encounters$ was already computed. Furthermore, instead of extending or constraining the EHS of an existing language, Aereal defines the whole EHS and makes it possible for developers to extended it or constrain it according to their needs.

Another important difference is that we use model checking techniques, instead of static analysis. This reflects the fact that we are dealing with exceptions in the earlier phases of development, where the implementation of the system is still not available. It is argued by some authors that designing the exceptional activity of a system since the early phases of development improves the overall system dependability [21].

Currently we are extending Aereal in order for it to support all the features of the proposed model. The first version of the framework used a simpler $Propagates$ relation and was based on a different set of assumptions. More specifically, it assumed that the $Signals$ and $Encounters$ relations were specified explicitly by framework users. Since specifying these relations by hand is a cumbersome and error-prone task, we are extending Aereal's transformation tool so that it can compute them automatically in terms of $Propagates$, $Handles$, and $Raises$ (which are specified by the user).

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, July 1997.
2. L. Bass et al. Air traffic control: A case study in designing for high availability. In *Software Architecture in Practice*, chapter 6. Addison-Wesley, 2nd edition, 2003.
3. R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE TSE*, SE-12(8):811–826, 1986.
4. F. Castor Filho et al. An architectural-level exception-handling system for component-based applications. In *Proceedings of the LADC'2003*, LNCS 2847, October 2003.
5. Fernando Castor Filho et al. A framework for analyzing exception flow in software architectures. In *Proceedings of WADS'2005*, May 2005. To appear.
6. Fernando Castor Filho et al. Modeling and analysis of architectural exceptions. Technical report, Institute of Computing - State University of Computing, 2005. To appear.
7. Paul C. Clements and Linda Northrop. Software architecture: An executive overview. Technical Report CMU/SEI-96-TR-003, SEI/CMU, February 1996.
8. Flaviu Cristian. *Dependability of Resilient Computers*, chapter 4- Exception Handling. BSP Professional Books, 1989.
9. M. Fahndrich et al. Tracking down exceptions in standard ml. Technical Report CSD-98-996, University of California, Berkeley, 1998.
10. A. Garcia et al. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software, Elsevier*, 59(2):197–222, 2001.
11. David Garlan et al. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, chapter 3. 2000.
12. James Gosling et al. *The Java Language Specification*. Addison-Wesley, 1996.
13. V. Issarny and J. P. Banatre. Architecture-based exception handling. In *Proceedings of the 34th HICSS*, 2001.
14. D. Jackson. Alloy: A lightweight object modeling notation. *ACM TOSEM*, 11(2), April 2002.
15. D. Jackson. Alloy home page, 2004. Address: `http://sdg.lcs.mit.edu/alloy/`.
16. Barbara Liskov and Alan Snyder. Exception handling in clu. *IEEE Transactions on Software Engineering*, pages 546–558, 1979.
17. D. Luckham et al. Specification and analysis of system architecture using rapide. *IEEE TSE*, 21(4):336–355, April 1995.
18. Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of FSE/ESEC'97*, September 1997.
19. M. Robillard and G. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM TOSEM*, 12(2):191–221, April 2003.
20. A. Romanovsky, P. Periorellis, and A. F. Zorzo. Structuring integrated web applications for fault tolerance. In *Proceedings of the 6th IEEE ISADS*, pages 99–106, 2003.
21. C. M. F. Rubira et al. Exception handling in the development of dependable component-based systems. *Software – Practice and Experience*, 35(5):195–236, March 2005.
22. C. F. Schaefer and G. N. Bundy. Static analysis of exception handling in ada. *Software: Practice and Experience*, 23(10):1157–1174, October 1993.
23. B. Schmerl and D. Garlan. Acmestudio: Supporting style-centered architecture development. In *Proceedings of the 26th ICSE*, May 2004.

Examining BPEL's Compensation Construct

Joey W Coleman

School of Computing Science
University of Newcastle upon Tyne
NE1 7RU, UK
email: `j.w.coleman@ncl.ac.uk`

Abstract. This paper gives a short description of some features of long-running transactions, as well as the language BPEL and its particular implementation of the compensation concept. Two examples are used to illustrate the application of BPEL's compensation construct. These examples, and reference to a structural operational semantics developed elsewhere, are used to help support an argument for the need of a more general implementation of compensation.

1 Introduction

Despite decades of work on ways of modelling long-running activities using transaction schemes, the same basic problems exist now as 25 years ago. Many systems have been designed to address parts of the problem but they tend to be refinements of the usual recovery mechanisms.

Designers of business process languages, in an attempt to model workflow, have taken to including a concept called compensation in their work. Compensation, in general, should be capable of addressing both non-reversible errors and non-erroneous changes in the execution of an activity. Unfortunately, the implemented design of compensations in languages such as BPEL can only conveniently be used to handle a subset of errors.

Possible semantic descriptions of BPEL's compensation mechanism are given in [Col04,BFN04] and others. The pragmatics of BPEL's compensation mechanism, on the other hand, need clarification. This paper aims to demonstrate how the mechanism's purpose as described in the BPEL specification [ACD^{+}03] is left unmet by the constraints given in the same document.

The next section of this paper gives a quick description of properties associated with the Long-Running Transactions (abbreviated as LRT). Section 3 describes BPEL and its compensation construct. Following that, section 4 gives an example that shows how BPEL's compensation model fits with a simple LRT. Section 5 extends that example into a scenario that does not easily fit BPEL's compensation model. The final section concludes this paper with a summary of the points raised by the examples and the motivation for a more general implementation of compensation.

2 Long-Running Transactions

The notion of long-running transactions [ACD$^+$03] arises out of work done in the database community on structuring transactions intended to run over long periods of time — from seconds up through minutes, hours, days and longer. In contrast, the well-studied notion of ACID transactions give desirable properties for transactions that run in short periods of time — nanoseconds, milliseconds, and up to a few seconds in length. The properties of consistency and durability are common to both LRTs and ACID transactions, but atomicity and isolation are very much weakened for LRTs [Gra81]. Synonyms for long-running transaction are long-lived transaction [Gra81,GMS87] and long-running activity [DHL91].

The properties of consistency and durability apply to LRTs in the same manner as they do with short-lived transactions. A LRT must leave the system in a consistent state, and more importantly, any changes made during the execution of the LRT that are visible outside of the LRT must also maintain system consistency. The durability requirement is obvious: having the changes made by a LRT that disappear except through the actions of another LRT is generally not a desirable thing.

Isolation can only be applied to those bits of state that are truly local to the LRT. Any state that does not survive past the end of the LRT should be isolated from everything outside of the LRT. Changes to the overall system state, however, cannot be isolated as the usual techniques used to isolate changes to the system state are unsuitable over long periods of time [GMS87].

Atomicity in the context of a LRT is very much relaxed when contrasted against its meaning for regular database transactions. For a LRT atomicity simply means that any changes during its progress maintain system consistency. The use of the compensation notion implicitly acknowledges the fact that there are cases where it is not possible to put the system state back to what it was before the start of the transaction.

As with short-lived transactions, LRTs have well-defined boundaries for their beginning and completion. They can, and usually should, contain short-lived transactions and even other LRTs.

3 BPEL's Compensation Construct

BPEL[1] [ACD$^+$03] is a relatively recent language that is still under development. Its origins lie in the web services community, and the initiators of its development include BEA Systems, IBM, Microsoft and a number of others. Current activity on the language is now coordinated by the OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee[2].

One of the claims made in the BPEL specification is that it provides the necessary tools and structure to support LRTs that are local to a BPEL process.

[1] Business Process Execution Language for Web Services
[2] Web address: http://www.oasis-open.org/

Central to that claim is BPEL's provision of a compensation construct, which was modelled after ideas in previous work on Sagas [GMS87] and others.

The specific implementation of compensation that BPEL uses is essentially an extension of the usual exception-handling mechanisms seen in languages such as `C++`, Java, and so on. Blocks of code — called scopes in BPEL — may have a compensation handler associated with them. These scopes, and their associated compensation handlers, can be nested to an arbitrary depth. Upon successful completion of the scope the compensation handler and the current state of the process are saved for possible later invocation.

Invocation of a compensation handler can only be done from within one of BPEL's fault (exception) handlers, and when actually invoked, the compensation handler is run on its associated saved state. It is not possible for the compensation handler to access the current state of the process directly, though it is not difficult to imagine a situation where current state is accessed by means of another process.

Three things characterize BPEL's compensation:

– the mechanism is intended to be a form of backward recovery [ACD$^+$03];
– despite saving the state of the process at scope completion, the mechanism only provides a convenient means to manipulate the process' control flow within the bounds of an exception handler;
– compensation handlers are named to facilitate control flow modification.

It is, in fact, possible to give an operational semantics that shows BPEL's compensation mechanism to be a primitive named procedure call [Col04]. Though the compensation "procedures" do not directly allow parameters, it could be argued that the saved state could be used as a parameter passing mechanism. The use of names to identify specific compensation handlers allows for an arbitrary, programmer-defined ordering, including parallel execution.

Categorizing BPEL's compensation feature as intended for backwards recovery comes directly from the BPEL specification [ACD$^+$03] which mentions the use of compensation to 'reverse' and 'undo' previous activities. The specification also goes so far as to restrict the invocation of a compensation handler to within a fault handler. Compensation in BPEL has been relegated to the realm of abnormal behaviour.

The assessment of BPEL's compensation mechanism as a convenient means to alter control flow relies on the fact that the language specification explicitly restricts compensation to only have meaning in a local sense. Saving the process state at scope completion is only intended to save the contents of the process' variables, *not* the underlying state of the BPEL processing engine [ACD$^+$03]. Saving those variables could be done manually and would give the compensation handler the added ability of being able to access the current state of the process. This leaves the single bit of control flow modification that BPEL's implementation of compensation usefully achieves: a simple mechanism to partition the actions of a traditional `try/catch`-style exception handler so that the handler need not try to figure out which parts of the body have executed.

4 The Bookshop

One of the common examples used to illustrate LRTs is that of a buyer-seller-shipper situation. Here we will consider a bookshop example similar to that used in [BFN04].

The example starts with the seller accepting an order for books in stock; an order for books not in stock is rejected immediately. Accepting the order reduces the seller's available inventory. The seller then attempts to fulfill the order by doing the following in parallel: a) arranging for the books to be shipped, b) packing the order, and c) checking the buyer's credit.

In this example we are not including the pickup of the books from the seller by the shipper, the receipt of the books by the buyer, the actual payment of the seller by the buyer, nor the payment of the shipper by the seller.

Included in the example are compensation actions for accepting the order, packing the order, and booking the shipper. Since checking the buyer's credit was just a 'read', there is no compensation required for that action. Of course, should the buyer's credit rating be insufficient for the order, then the order will be canceled.

It is straightforward to cancel this LRT at any point during its execution. If the seller in the LRT had only just completed accepting the order, canceling it involves merely throwing the order away, making the books available for sale, and notifying the buyer that the order was canceled. If the order had been accepted and the parallel tasks were in progress, then canceling involves unbooking the shipper and unpacking any packed books, then throwing away the order and making the books available for sale.

This example translates into a BPEL process in a straightforward manner. Accepting an order would exist as a BPEL `scope` object (with its compensation handler). Unbooking the shipper would also be in its own `scope` object. However, to correctly model the required compensation for the parallel tasks, each action that packs a book would need to be in its own `scope`, thus allowing only the compensation handlers for the packed books to run.

5 The Bookshop, Extended

The previous example is fairly straightforward, and perhaps even matches the most common behaviour that a bookseller might follow. There is, however, a more complex behaviour that shows the limitations of BPEL's compensation model.

For this example, imagine that the seller carries no stock, such that all books must be pre-ordered. In this case, the seller accepts any order for any collection of books that it believes it can get. For the sake of simplicity, the seller also knows the correct final price for any book that it believes to be available.

The seller would then generate, in parallel, an expected delivery time for the buyer, and charge the buyer for the books. This charge may simply be a deposit or the full cost of the book, but it would be a charge rather than a credit check. After the buyer has been charged, the seller then places an order with their supplier for the desired books. When the seller receives the books it would perform the parallel tasks of arranging for the books to be shipped and packing all of the books to be shipped to the client.

If the seller's suppliers were completely reliable, then the mechanism for canceling this order is a straightforward extension of the previous example. Since it is unlikely that the suppliers would be completely reliable, we will assume that suppliers will occasionally be unable to supply certain books, and that the suppliers will notify the seller of this when the seller places their order from the supplier.

Assume, instead, at some point between when the seller told the buyer when to expect the books and when the seller should have received the books, that the seller receives some notification that one of the books in the order is no longer available. This requires that things be corrected so that the unavailable book is no longer a part of the order.

If there was only one book in the order then handling this situation is easy: just cancel the whole LRT. For cases where there were several books in the order, especially when the seller has already received some of the books, then the seller will still need to ship the rest of the books and record things appropriately.

Simply canceling then restarting the whole LRT without the unavailable book is inappropriate: that would likely cost the seller extra in transaction fees. It is also an unnecessary repetition of effort. What should happen is that the unavailable book is removed from the order, the buyer is refunded the appropriate amount, and then the LRT proceeds as though the unavailable book had not been ordered in the first place.

Having the supplier's notification about the unavailability of a book forces the seller to perform compensatory actions so as to avoid having to cancel the entire LRT. It would seem appropriate to use a compensation mechanism to allow the LRT to proceed, but the actual implementation in BPEL would be absurdly complex.

Despite BPEL's restriction that compensation may only be called from within fault handlers, it is possible to model this behaviour using compensation. The design would have each book ordered in its own thread — as with the process given above — but after the individual book order is complete, a busy-wait loop would prevent the thread and associated compensation scopes from exiting. If one of the book orders needs to be compensated, then a flag would be set and the busy-wait loop would throw a fault which in turn would cause a handler to invoke the compensators.

The problem with this solution — aside from its inelegance — is that it seems to run against the pragmatics of a compensation construct to use nested scopes just to isolate fault handlers whose only purpose is to invoke a compensator for the innermost scope.

So why don't we just use the fault handler to directly correct the problem? Leaving aside the convenience of a mechanism that automatically only corrects the actions that have completed, this solution is not much better. The busy-wait loops are still required, leaving a collection of live threads that are doing nothing. Compare this against a compensation mechanism where, when the compensator is installed, the individual threads are finished and cleaned up normally.

6 Conclusions

From the standpoint of the structural operational semantics mentioned above and developed in [Col04], BPEL's compensation mechanism is only a primitive procedure call. Indeed, the compensation mechanism doesn't even require the full structure of a procedure call, but just that of a block-structure language. The only restriction in the semantic model that prevents the compensation mechanism from from being used outside of a fault handler is a well-formedness condition on the abstract syntax of the language. In light of this it is unfortunate that the BPEL specification states this condition as it needlessly precludes a more general use of compensation.

The initial example does show that the compensation model used in BPEL is applicable to some situations. The use of named compensators has advantages, allowing programmer-defined compensation ordering. Also, in general, the usual implementations of compensation are extremely useful to simplify the programmer's task of only reversing those actions that have completed successfully.

The use of compensation as implemented in BPEL to handle changes to the LRT during its execution is inconvenient at best. An argument can be raised that any changes to the LRT should be kept well defined and incorporated into the main flow of the process. This argument has the same problems as arguing that fault handling should be done inline with the main code rather than separated out into fault handlers.

Some models of compensation have posited a fairly strong property: if an action and its compensation are independent of all of the actions between the original action and the compensation, then the action and its compensation is equivalent to a null action [BHF04]. While this is certainly true, it has been pointed out that the likelihood of independence is rather low [GFJK03]. First, there is the difficulty of designing your process so that the interleaving of local actions maintains this independence. Second, due to the long-running nature of LRTs, and the requisite lack of lock-based synchronization, it is extremely likely that another executing LRT will do something that is not completely independent. One might observe that part of the point of grouping actions together in a transaction is because those actions are not independent.

The extended example gives an argument that the particular compensation model used in BPEL is not applicable to the full range of situations where compensation would seem to be an ideal tool to structure a long-running transaction's fault tolerance. If BPEL's compensation mechanism could be invoked outside of a fault handler then the extended bookshop example would no longer be an issue.

Acknowledgments
The author would like to acknowledge the support of the RODIN (Rigorous

Open Development Environment for Complex Systems) Project funded by the European IST and the DIRC (Dependability Interdisciplinary Research Collaboration) project funded by EPSRC/UK.

References

[ACD$^+$03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Language for Web Services, version 1.1. http://www.ibm.com/developerworks/webservices/library/ws-bpel/, May 2003.

[BFN04] Michael Butler, Carla Ferreira, and Muan Yong Ng. Precise modelling of compensating business transactions and its application to BPEL. Technical report, University of Southampton, Electronics and Computer Science, 2004.

[BHF04] Michael Butler, Tony Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In A. Abdallah and J. Sanders, editors, *Proceedings of 25 Years of CSP (in press)*, London, 2004.

[Col04] Joseph W Coleman. Features of BPEL modelled via structural operational semantics. MPhil thesis, University of Newcastle Upon Tyne, November 2004.

[DHL91] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. A transactional model for long-running activities. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 113–122. Morgan Kaufmann Publishers Inc., 1991.

[GFJK03] Paul Greenfield, Alan Fekete, Julian Jang, and Dean Kuo. Compensation is not enough. In *EDOC '03: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*, page 232. IEEE Computer Society, 2003.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM Press, 1987.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations. In *VLDB*, pages 144–154. IEEE Computer Society, September 1981.

On Specification and Verification of Location-based Fault Tolerant Mobile Systems

Alexei Iliasov, Victor Khomenko, Maciej Koutny and Alexander Romanovsky

School of Computing Science, University of Newcastle
Newcastle upon Tyne, NE1 7RU, United Kingdom

Abstract. In this paper, we investigate context aware location-based mobile systems. In particular, we are interested how their behaviour, including fault tolerant aspects, could be captured using a formal semantics amenable to rigorous analysis and verification. We propose a new formalism and middleware called Cama, which provides a rich environment to test our approach. The approach itself aims at giving Cama a concurrency semantics in terms of a suitable process algebra, and then applying efficient model checking techniques to the resulting process expressions in a way which alleviates the state space explosion. The model checking technique adopted in our work is partial order model checking based on Petri net unfoldings, and we use a semantics preserving translation from the process terms used in the modelling of Cama to a suitable class of high-level Petri nets.

1 Introduction

Mobile agent systems are increasingly attracting attention of software engineers. However, issues related to fault tolerance and exception handling in such systems have not received yet the level of attention they deserve. In particular, formal support for validating the correctness and robustness of fault tolerance properties is still under-developed. In this paper, we will outline the initial steps of our approach to dealing with such issues in the context of a concrete system for dealing with mobility of agents (Cama), and a concrete technique for verifying their properties (partial order model checking). Our aim in this paper is to present a formal model for the specification, analysis and model checking of Cama designs. In doing so, we will use process algebras and Petri nets.

In concrete terms, our approach is first to give a formal semantics (including a compositional translation) of a suitably expressive subset of Cama in terms of an appropriate process algebra and its associated operational semantics. The reason why we chose a process algebra semantics is twofold: (i) process algebras, due to their compositional and textual nature, are very close to the actual notations and languages used in real implementations; and (ii) there exists a significant body of research on the analysis and verification of process algebras. In our particular case, there are two process algebras which are directly relevant to Cama, viz. Klaim [2] and π-calculus [9], and our intention is to use the former as a starting point for the development of the formal semantics.

2 Location-based fault tolerant mobile systems

The design of our system has been strongly influenced by LINDA [6], which is a set of language-independent coordination primitives that can be used for communication and coordination between several independent pieces of software. Thanks to its language independence, LINDA has become quite popular, and many programming languages have one or more implementations of its coordination primitives. Coordination primitives presented in LINDA allow processes to put, get and test for tuples in a tuple space shared by the running processes. A tuple is a vector of typed data values some of which can be empty (in which case they match any value of a given type). Certain operations, such as *get* and *test*, can be blocking. This provides effective inter-process coordination; other kinds of coordination primitives, such as semaphores, can be readily simulated.

We will use an *asymmetric* communication scheme which is closer to the traditional service provision architectures. It is based on the concept of a fairly reliable infrastructure-provided wireless connectivity. (The alternative *symmetric* scheme can also operate in ad-hoc networks and all the coordination functionality is implemented by the agents.) In the asymmetric scheme, the larger part of the coordination logic is moved to a location server. This approach is able to support large-scale mobile agent networks in a predictable and reliable manner. It makes better use of the available resources since most of the operations are executed locally. Moreover, the asymmetric architecture eliminates the need for employing complex distributed algorithms or any kind of remote access. This allows us to guarantee atomicity of certain operations without sacrificing performance and usability. Another advantage is that it provides a natural way of introducing context-aware computing by defining location as a context. The main disadvantage of the location-based scheme is that an additional infrastructure is always required to support mobile agent collaboration.

A CAMA (context-aware mobile agents) system consists of a set of *locations*, and active entities of the system, called *agents*. An agent is a piece of software which is executed on its own *platform*, providing execution environment interface to the location middleware. Agents can only communicate with other agents in the same location. Agents can migrate logically (connection and disconnection) or physically (e.g., movement of a PDA on which the agent is hosted on) from a location to a location. Agents can also migrate logically from platform to platform using weak code mobility (transfer of application code or its parts from one host to another without retaining the execution state). Compatible agents (i.e., agents capable of cooperation in certain conditions in order to achieve individual agent goals and in accordance to the abstract specification of the whole system) collaborate through a scoping mechanism, where a *scope* defines a joint activity of several agents. Scoping mechanism also isolates non-compatible agents from each other. More details about the introduced concepts are provided below.

Scope is a dynamic container for tuples. It provides an isolated coordination space for compatible agents, by restricting the visibility of tuples contained within the scope to the participants of the scope. A scope is initiated by an agent and then atomically created by a location when all the participating agents are ready. It is defined by the set of roles, a minimal required number of active roles, and a maximal allowed number of active roles. Scopes can be nested as scope participants can create new contained scopes.

Role is an abstract description of agent functionality. Each role is associated with some abstract scope model. Agent may implement a number of roles and can also play several roles within the same scope or different scopes. There is a formal relationship between a scope and its role. The latter is formally derived from an abstract model through decomposition process, while the former is a run-time instantiation of such an abstract model as it is formed via a composition of agent roles (for more discussion see [7]).

Location is a container for scopes. It can be associated with a particular physical location and can have certain restrictions on the types of supported scopes. It is the core part of the system as it provides means of communication and coordination between agents. We may assume that each location has a unique name. This roughly corresponds to IP addresses of hosts in a network which are often unique in some local sense. A location must keep track of the agents present and their properties in order to be able to automatically create new scopes and restrict access to the existing ones. Locations may provide additional services that can vary from location to location. These are made available to agents via what appears as a normal scope though some roles are implemented by the location system software. As with all the scopes, agents are required to implement specific roles in order to connect to a location-provided scope. Examples of such services include printing on a local printer, Internet access, making a backup to a location storage, and migration. In addition to supporting scopes as means of agent communication, locations may also support logical mobility of agents, hosting of platforms and agent backup. Hosting of platform on a location allows an agent to run without a support from, say, a PDA. For example, a user may decide to move an agent from the PDA to a location before leaving the location. When requested by an agent, a location may play in certain types of scopes the role of a trusted third party that is neutral to all the participating agents. This facilitates implementation of various transaction schemes.

Platform provides an execution environment for an agent. It is composed of a virtual machine for code execution, networking support, and middleware for interacting with a location. A platform may be supported by a PDA, smartphone, laptop or a location server. The notion of a platform is important to clearly differentiate between the concept of a location providing coordination services to agents, and the middleware that only supports agent execution. In other approaches no such distinction is usually made [10, 3, 11].

Agent is a piece of software implementing a set of roles which allow it to take part in certain scopes. All agents must implement some minimal functionality, called the default role, which specifies their activities outside of all the scopes.

3 A process algebra for CAMA systems

The semantical model of CAMA will be captured using a process algebra based on KLAIM [2] and also the π-calculus [9]. We now briefly outline some key aspects of this development (see [5] for details).

We assume that $\mathcal{L}$ is a set of *localities* ranged over by $l, l', l_1, \ldots$ and a disjoint set $\mathcal{U}$ of *locality variables* ranged over by $u, v, w, u', v', w', u_1, v_1, w_1, \ldots$. (We also assume that a special locality `self` belongs to $\mathcal{L}$.) Their union forms the set of *names* ranged over by $\ell, \ell', \ell_1, \ldots$. In addition, $\mathcal{A} = \{A_1, \ldots, A_m\}$ is a finite set of *process identifiers*, each identifier $A \in \mathcal{A}$ having a finite arity n_A.

The syntax comes in four parts: networks, actions, processes and templates.

$$\begin{array}{lcll}
N & ::= & l :: P \;\mid\; l :: \langle l \rangle \;\mid\; N \,\|\, N & \text{(networks)} \\
a & ::= & \mathbf{out}(\ell)@\ell \;\mid\; \mathbf{in}(T)@\ell \;\mid\; \mathbf{eval}(A(\ell_1, \ldots, \ell_{n_A}))@\ell & \text{(actions)} \\
P & ::= & \mathbf{nil} \;\mid\; A(\ell_1, \ldots, \ell_{n_A}) \;\mid\; a \,.\, P \;\mid\; P + P \;\mid\; P|P & \text{(processes)} \\
T & ::= & \ell \;\mid\; !z & \text{(templates)}
\end{array}$$

Moreover, for each $A \in \mathcal{A}$, there is exactly one definition $A(u_1, \ldots, u_{n_A}) \overset{\text{df}}{=} P_A$, which is available across the whole network.

Networks are finite collections of computational nodes, where data and processes can be located. Each node consists of a locality l identifying it and a process or a datum (itself a locality in this simple presentation). There can be several nodes with the same locality part. Effectively, one may think of a network as a collection of uniquely named nodes, each node comprising its own data space and a possibly concurrent process which runs there (for simplicity, we assume that only *singleton* tuples are stored). This view is embodied in the rules for *structural equivalence* on nodes and networks, such as $N_1 \,\|\, N_2 \equiv N_1 \,\|\, N_2$, $(N_1 \,\|\, N_2) \,\|\, N_3 \equiv N_1 \,\|\, (N_2 \,\|\, N_3)$ and $l :: (P_1|P_2) \equiv l :: P_1 \,\|\, l :: P_2$.

Actions are the basic (atomic) operations which can be executed by processes, as follows: $\mathbf{out}(\ell')@\ell$ deposits a fresh copy of ℓ' inside the locality addressed by ℓ; $\mathbf{in}(T)@\ell$ retrieves an item matching the template T from the locality addressed by ℓ; and $\mathbf{eval}(A(\ell_1, \ldots, \ell_{n_A}))@\ell$ instantiates a new copy of the process identified by A in the locality addressed by ℓ.

Processes act upon the data stored at various nodes and spawn new processes. The algebra of processes is built upon the (terminated) process **nil** and three composition operators: prefixing by an action $(a \,.\, P)$; choice $(P_1 + P_2)$; and parallel composition $(P_1|P_2)$.

The action prefix $\mathbf{in}(!z)@\ell \,.\, P$ *binds* the locality variable z within P, and we denote by $fn(P)$ the *free* names of P (and similarly for networks). For the process definition, we assume that $fn(P_A) \subseteq \{u_1, \ldots, u_{n_A}\}$. Processes are defined up to the *alpha-conversion*, and $\{\ell/\ell', \ldots\}P$ will denote the agent obtained from P by replacing all free occurrences of ℓ' by ℓ, etc, possibly after alpha-converting P in order to avoid name clashes. We assume that a network is *well-formed*, i.e., no name across the network and process definitions is both free and bound, it never generates more than one binding, and there are no free locality variables.

The operational semantics of networks and processes is based on the structural equivalence $\equiv$ and labelled transition rules providing the record of an execution, e.g., output and input involve the following SOS rules:

$$\frac{\text{if } \ell = \texttt{self} \text{ then } l'' = l \text{ else } l'' = \ell}{l :: \mathbf{out}(\ell)@l' \,.\, P \xrightarrow{\mathtt{o}(l,l'',l')} l :: P \parallel l' :: \langle l'' \rangle}$$

$$l :: \mathbf{in}(!z)@l' \,.\, P \parallel l' :: \langle l'' \rangle \xrightarrow{\mathtt{i}(l,l'',l')} l :: \{l''/z\}P \parallel l' :: \mathbf{nil}$$

The semantics of Cama operations is given using a straightforward extension of the process algebra outlined above.

3.1 Process algebra semantics of CAMA

The basic parts of the Cama system are locations, scopes, agents and middleware. Locations provide scopes which, in turn, provide a private coordination space to communicating agents. Middleware is an active entity that controls the state of a location and provides certain services, such as scope creation. Agents can synchronise using Linda-style operations on scopes. Scopes can contain sub-scopes thus providing a hierarchy of nested agent activities. The subset of the Cama operations chosen for model-checking comprises a number of location/scope operations:

$EngageLocation$	$DisengageLocation$	$CreateScope$	$GetScopes$
$DeleteScope$	$JoinScope$	$LeaveScope$	

and a number of synchronisation operations: **in**, **rd**, **inp**, **rdp**, **out**, **ina**, **rda**, **inpa** and **rdpa**. All these operations require locality variable argument which is a reference to a location. In Cama, locations are static and hence they never appear or disappear during an agent's lifetime (dynamic locations creation and destruction can be simulated by other means). Operations occurring within a locality l are denoted as, e.g., $\mathbf{eval}()@l$. Synchronisation primitives take a scope name instead of a location, and we assume that location names are contained within the scope names. For brevity, the locality l may be omitted if its value is clear from the context. To model nested scopes, we use the notion of a location tuple prefix, corresponds to one or more fields of a tuple. The syntax of tuple prefixes $\mathfrak{p}$ is based on that of tuple/template:

$$t \ ::= \quad ? \ \mid \ !z \ \mid \ t, t$$

where '?' is a wildcard matching any field value, and 't_1, t_2' is field concatenation. We than define $\mathfrak{p} = \langle t \rangle$ as well as use '$*$' for prefix concatenation, $\mathfrak{p}^n$ for prefix repetition, and $\mathfrak{p}*$ for an *open* prefix. We also use the following operations:

– $[\mathfrak{p}](n)$ is the value of the n-th field of a tuple with the prefix $\mathfrak{p}$ where field count starts after the prefix part. For example, $[\mathfrak{a}](2)$ applied to a tuple space containing $\mathfrak{a} * \langle a_1, a_2 \rangle$ can give a_2 (note that matching is non-deterministic if $\mathfrak{p}$ is a prefix of more then one tuple).

– $[\mathfrak{p}]'(n)$ is the same as $[\mathfrak{p}](n)$ but it also removes the matched tuple.

– $[\mathfrak{p}(n)]$ is the bag of values of the n-th fields of all $\mathfrak{p}$-matching tuples.

All these operations assume that there is at least one tuple matching $\mathfrak{p}$ and the length of any tuple that can be matched is at least equal to the length of $\mathfrak{p}$ plus n, otherwise operation's behaviour is undefined. Note that it is possible to express the above operations via standard LINDA constructs; for example, assigning $[\mathfrak{p}](n)$ to a variable v is equivalent to $\mathbf{rd}(\mathfrak{p} * \langle ?\rangle^n * \langle !v\rangle)$. Finally, the open prefix matches all the tuples starting with a given prefix, and so tuples of different length and structure may be matched.

To model scopes and the location middleware behaviour, we need a structuring of tuple space through special prefixes, as given in the table below:

Prefix name	Description
$\mathfrak{m}*$	Requests to the middleware
$\mathfrak{i}*$	Possible agent names
$\mathfrak{a}*$	Issued agent names
$\mathfrak{s}*$	Scopes
$\mathfrak{s} * s*$	Description structures of scope s
$\mathfrak{s} * s * \mathfrak{r}*$	Roles of a scope
$\mathfrak{s} * s * \mathfrak{n}*$	Number of roles in a scope
$\mathfrak{s} * s * \mathfrak{r} * r * \langle min, max\rangle$	Restrictions on individual role r
$\mathfrak{s} * s * \mathfrak{d}*$	Dynamic state of a scope instance
$\mathfrak{s} * s * \mathfrak{c} *$	Contents of scope s

We need two auxiliary operations, $\mathbf{lock}(\mathfrak{p})$ and $\mathbf{unlock}(\mathfrak{p})$, which grant and release exclusive access to all the tuples beginning with a prefix $\mathfrak{p}$:

$$\begin{aligned} \mathbf{lock}(\mathfrak{p}) &\overset{\text{df}}{=} \mathbf{in}(\mathfrak{X} * \mathfrak{p} * \langle 1\rangle)\,.\,\mathbf{out}(\mathfrak{X} * \mathfrak{p} * \langle 0\rangle) \\ \mathbf{unlock}(\mathfrak{p}) &\overset{\text{df}}{=} \mathbf{in}(\mathfrak{X} * \mathfrak{p} * \langle 0\rangle)\,.\,\mathbf{out}(\mathfrak{X} * \mathfrak{p} * \langle 1\rangle) \end{aligned}$$

Many operations are carried out by the location middleware, which is modelled as a set of looped event handlers waiting for certain tuples with prefix $\mathfrak{m}$ to appear. A middleware process $P_{mid}@l$ is defined as parallel composition of the event handling processes: $P_{EngageLocation}$, $P_{DisenageLocation}$, $P_{CreateScope}$, $P_{DeleteScope}$, $P_{JoinScope}$, $P_{LeaveScope}$, $P_{ScopeActivate}$ and $P_{ScopeDeactivate}$. In each case, there is an agent side code that sends a request and collects any returned data, using some additional operations, such as $AEngageLocation@l$ and $ADisengageLocation@l$.

Engage location operation registers an agent in a given location and issues a new name that is guaranteed to be location-wide unique; it allows the agent to execute other operations in the location. This operation is always the first one executed by an agent when it connects to a new location.

$$\begin{aligned} \mathit{AEngageLocation}@l &\overset{\mathrm{df}}{=} \mathbf{lock}(\mathfrak{m})\,.\,\mathbf{out}(\mathfrak{m} * \langle \textsc{engage}\rangle)\,.\\ &\ \mathbf{in}(\mathfrak{e} * \langle !a\rangle)\,.\,\mathbf{unlock}(\mathfrak{m}) \end{aligned}$$

$$\begin{aligned} P_{\mathit{EngageLocation}} &\overset{\mathrm{df}}{=} \mathbf{in}(\mathfrak{m} * \langle \textsc{engage}\rangle)\,.\,\mathbf{in}(\mathfrak{i} * \langle !a\rangle)\,.\\ &\ \mathbf{out}(\mathfrak{a} * \langle a\rangle)\,.\,\mathbf{out}(\mathfrak{e} * \langle a\rangle)\,.\\ &\ P_{\mathit{EngageLocation}}(N) \end{aligned}$$

Disengage location removes the registered agent name from the internal registry of the agent names.

$$\begin{aligned} \mathit{ADisengageLocation}@l &\overset{\mathrm{df}}{=} \mathbf{out}(\mathfrak{m} * \langle \textsc{disengage}, a\rangle) \end{aligned}$$

$$\begin{aligned} P_{\mathit{DisenageLocation}} &\overset{\mathrm{df}}{=} \mathbf{in}(\mathfrak{m} * \langle \textsc{disengage}, !a\rangle)\,.\\ &\ \mathbf{in}(\mathfrak{a} * \langle a\rangle)\,.\,P_{\mathit{DisenageLocation}} \end{aligned}$$

Create scope adds a new scope defined by a name and a special record d that describes the scope structure and the role that the creating agent will assume. The record d has the following fields: *rolesn* - the number of roles, *roles* - the vector of role names, *min* - the minimal required participants number, and *max* - the maximum allowed participants number.

$$\mathit{ACreateScope}(a, s, d, r)@l \overset{\mathrm{df}}{=} \mathbf{out}(\mathfrak{m} * \langle \textsc{create\_scope}, a, s, d, r\rangle)$$

```
$P_{CreateScope} \overset{\mathrm{df}}{=} \mathbf{in}(\mathfrak{m} * \langle \textsc{create\_scope}, !a, !s, !d, !r\rangle)\,.\,\mathbf{lock}(\mathfrak{s})\,.$
    if$(a \in [\mathfrak{a}(1)] \quad \wedge \quad s \in [\mathfrak{s}(1)] \quad \wedge \quad r \in d.roles)$
    then
        $\mathbf{out}(\mathfrak{s} * s * \mathfrak{n} * \langle d.rolesn\rangle)\,.\,\mathbf{outa}(\mathfrak{s} * s * \mathfrak{r} * \langle d.roles\rangle)\,.$
        $\mathbf{outa}(\mathfrak{s} * s * \mathfrak{r} * \langle d.roles, d.min, d.max\rangle)\,.$
        $\mathbf{outa}(\mathfrak{s} * s * \mathfrak{d} * \langle d.roles, 0\rangle)$
    endif$\,.\,\mathbf{in}(\mathfrak{s} * s * \mathfrak{d} * \langle r, 0\rangle)\,.\,\mathbf{out}(\mathfrak{s} * s * \mathfrak{d} * \langle r, 1\rangle)\,.$
    $\mathbf{out}(\mathfrak{s} * s * \mathfrak{c} * \langle a\rangle)\,.\,\mathbf{out}(\mathfrak{m} * \langle \textsc{activator}, s\rangle)\,.$
    $\mathbf{out}(\mathfrak{e} * \langle \textsc{join}, s\rangle)\,.\,\mathbf{unlock}(\mathfrak{s})\,.\,P_{CreateScope}$
```

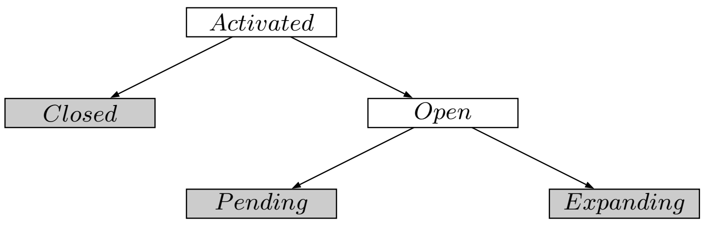


Fig. 1. Hierarchy of scope states.

A scope becomes *activated* after some agent creates it with the $CreateScope$ operation. Scope is *open* when there are some vacant roles in it. Scope is *closed* when all the roles are taken. Scope is *pending* if some required roles are not taken yet and *expanding* if all the required roles are taken but there still some vacant roles (see figure 1).
Delete scope destroys a scope which must be owned by the calling agent. Any contained scopes are also destroyed.

$$ADeleteScope(a, s)@l \overset{\mathrm{df}}{=} \mathbf{out}(\mathfrak{m} * \langle \text{DELETE\_SCOPE}, a, s\rangle)$$

The middleware process simply removes all the tuples associated with the scope and any of its sub-scopes.

$$\begin{aligned} P_{DeleteScope} \overset{\mathrm{df}}{=} {} & \mathbf{lock}(\mathfrak{m}) \,.\, \mathbf{in}(\mathfrak{m} * \langle \text{DELETE\_SCOPE}, !a, !s\rangle) \,.\, \mathbf{inpa}(\mathfrak{s} * s*) \,.\\ & \mathbf{inpa}(\mathfrak{d} * s*) \,.\, \mathbf{unlock}(\mathfrak{m}) \,.\, P_{DeleteScope} \end{aligned}$$

Join scope puts an agent into an existing scope if there is appropriate vacant role in the scope.

$$AJoinScope(a, s, r)@l \overset{\mathrm{df}}{=} \mathbf{out}(\mathfrak{m} * \langle \text{JOIN\_SCOPE}, a, s, r\rangle)$$

This operation may trigger scope activation or change of the state from open to closed. The middleware process adds new participant to the scope and announces the event.

```
$P_{JoinScope} \overset{\mathrm{df}}{=} \mathbf{in}(\mathfrak{m} * \langle \text{JOIN\_SCOPE}, !a, !s, !r\rangle)$ .
    $\mathbf{lock}(\mathfrak{s})$ .
    if  $(a \in [\mathfrak{a}(1)] \;\wedge\; s \in [\mathfrak{s}(1)] \;\wedge\; r \in [\mathfrak{s} * s * \mathfrak{r}(1)] \;\wedge$
          $[\mathfrak{s} * s * \mathfrak{d} * \mathfrak{r} * r](1) < [\mathfrak{s} * s * \mathfrak{r} * r](2))$
    then
        $\mathbf{out}(\mathfrak{s} * s * \mathfrak{c} * \langle a\rangle)$ .
        $\mathbf{out}(\mathfrak{s} * s * \mathfrak{d} * \mathfrak{r} * \langle r, [\mathfrak{s} * s * \mathfrak{d} * \mathfrak{r} * r]'(1) + 1\rangle)$ .
        $\mathbf{out}(\mathfrak{e} * \langle \text{JOIN}, s\rangle)$
    endif . $\mathbf{unlock}(\mathfrak{s})$ . $P_{JoinScope}$
```

Leave scope removes the calling agent from a given scope and role.

$$ALeaveScope(a, s, r)@l \overset{\mathrm{df}}{=} \mathbf{out}(\mathfrak{m} * \langle \text{LEAVE\_SCOPE}, a, s, r\rangle)$$

The middleware process removes record about the agent and issues an event that may trigger scope state update.

```
$P_{LeaveScope} \overset{\mathrm{df}}{=} \mathbf{in}(\mathfrak{m} * \langle \text{LEAVE\_SCOPE}, !a, !s, !r\rangle)$ . $\mathbf{lock}(\mathfrak{s})$ .
    if  $a \in [\mathfrak{s} * s * \mathfrak{c}(1)]$
    then
      $\mathbf{out}(\mathfrak{s} * s * \mathfrak{d} * \mathfrak{r} * \langle r, [\mathfrak{s} * s * \mathfrak{d} * \mathfrak{r} * r]'(1) - 1\rangle)$
      $\mathbf{in}(\mathfrak{s} * s * \mathfrak{c} * \langle a\rangle)$ . $\mathbf{out}(\mathfrak{e} * \langle \text{LEAVE}, s\rangle)$
    endif . $\mathbf{unlock}(\mathfrak{s})$ . $P_{LeaveScope}$
```

LINDA operations that we are using also sugared with additional checks for a scope's state:

- $\mathbf{in}(t)@s \overset{\mathrm{df}}{=} \mathbf{rd}(\mathfrak{s}*s*\langle\textsc{ready}\rangle)\,\boldsymbol{.}\,\mathbf{in}(\mathfrak{s}*s*\mathfrak{c}*t)$ removes and returns a tuple that matches the supplied tuple template. First it checks if the specified scope exists and that it is ready. If it not so the operation blocks until conditions change. When there is no tuple available immediately it also blocks until one appears. In case of multiple matching tuples the result is chosen non-deterministically.
- $\mathbf{out}(t)@s \overset{\mathrm{df}}{=} \mathbf{rd}(\mathfrak{s}*s*\langle\textsc{ready}\rangle)\,\boldsymbol{.}\,\mathbf{out}(\mathfrak{s}*s*\mathfrak{c}*t)$ outputs a tuple into a scope. First it checks if the target scope is available and ready.

Other operations are defined in a similar manner. Each operation is prefixed by $\mathbf{rd}(\mathfrak{s}*s*\langle\textsc{ready}\rangle)$ and a tuple or template argument is prefixed with the prefix corresponding to the scope. Operations acting on vector of tuples can be expressed via other operation using prefix locking function.

Whenever a *join* event occurs (meaning a joining of an agent to a scope), the scope activate process checks if the state of the scope in question need to be updated. There are two possible situations. The first one is when all the required roles are fulfilled and the scope changes its state from *pending* to *ready*. As a result, the process issues a tuple that triggers execution of possibly suspended earlier Linda operations. Another situation is when all the possible roles are taken and no more agents should be able to connect to this scope. In this case the scope becomes *closed* and this prevents any other agents from entering it.

$$
\begin{array}{rl}
P_{ScopeActivate} \overset{\mathrm{df}}{=} & \mathbf{in}(\mathfrak{e}*\langle\textsc{Join},!s\rangle)\,\boldsymbol{.}\\
& \texttt{if}\ \ (s\in[\mathfrak{s}(1)]\ \ \wedge\\
& \qquad \forall\rho\in[\mathfrak{s}*s*\mathfrak{r}(1)] : [\mathfrak{s}*s*\mathfrak{d}*\rho](1)\geq[\mathfrak{s}*s*\mathfrak{r}*\rho](1))\\
& \texttt{then}\ \ \mathbf{in}(\mathfrak{s}*s*\langle!\textsc{state}\rangle)\,\boldsymbol{.}\,\mathbf{out}(\mathfrak{s}*s*\langle\textsc{ready}\rangle)\,\boldsymbol{.}\\
& \texttt{if}\ \ (s\in[\mathfrak{s}(1)]\ \ \wedge\\
& \qquad \forall\rho\in[\mathfrak{s}*s*\mathfrak{r}(1)] : [\mathfrak{s}*s*\mathfrak{d}*\rho](1)=[\mathfrak{s}*s*\mathfrak{r}*\rho](2))\\
& \texttt{then}\ \ \mathbf{in}(\mathfrak{s}*s*\langle!\textsc{state}\rangle)\,\boldsymbol{.}\,\mathbf{out}(\mathfrak{s}*s*\langle\textsc{closed}\rangle)\,\boldsymbol{.}\\
& P_{ScopeActivate}
\end{array}
$$

Moreover, $P_{ScopeDeactivate}$ updates the state of a scope whenever some agent leaves it.

4 Model checking CAMA systems

Mobile systems are highly concurrent causing a state space explosion when applying model checking techniques. We therefore use approach which copes well with such a problem based on partial order semantics of concurrency and the corresponding Petri net unfoldings.

A *finite and complete unfolding prefix* of a Petri net PN is a finite acyclic net which implicitly represents all the reachable states of PN together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* PN, by successive firings of transition, under the following assumptions: (i) for each new firing a fresh transition (called an *event*) is generated; (ii) for each newly produced token a fresh place (called a *condition*) is generated. If PN has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix.

Efficient algorithms exist for building such prefixes [8], and complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, then the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will be isomorphic to the net itself. Since mobile systems usually exhibit a lot of concurrency, their unfolding prefixes are often much more compact than the corresponding state graphs. Therefore, unfolding prefixes are well-suited for alleviating the state space explosion problem. To apply net unfoldings, we need to translate process algebra terms corresponding to CAMA systems into Petri nets.

4.1 From process algebra to Petri nets

The development of Petri net model corresponding to expressions of the process algebra for CAMA systems has been inspired by the box algebra [1] and by the rp-net algebra used in [4] to model π-calculus. It uses coloured tokens and read-arcs (allowing any number of transitions to simultaneously check for the presence of a resource stored in a place). Transitions can have different labels, such as **o** to specify outputting of data to tuple spaces, **i** to specify retrieving of data from tuple spaces, and **e** to specify process creation.

A key idea behind the translation is to view a system as consisting of a main program together with a number of procedure declarations. We then represent the control structure of the main program and the procedures using disjoint unmarked nets, one for the main program and one for each of the procedure declarations. The program is executed once, while each procedure can be invoked several times (even concurrently), each such invocation being uniquely identified by structured tokens which correspond to the sequence of recursive calls along the execution path leading to that invocation. With this in mind, we use the notion of a *trail* σ to denote a finite (possibly empty) sequence of **e**-labelled transitions. And the places of the nets which are responsible for control flow will carry tokens which are simply trails. (The empty trail will be treated as the usual 'black' token.) Procedure invocation is then possible if each of the input places of a transition t labelled with **e** contains the same trail token σ, and it results in removing these tokens and inserting a new token σt in each initial (entry) place of the net corresponding to the definition of $A(\ldots)$, together with other tokens representing the corresponding actual parameters. Places are labelled in ways reflecting their intended role, as explained below.

– *Control flow places:* These will be used to model control flow and be labelled by their status symbols (*internal* places by i, and *interface* places by e and x, for entry and exit, respectively).

- *Locality places (or loc-places):* These will be labelled by localities in $\mathcal{L}$ and carry structured tokens representing localities known and used by the main program and different procedure invocations. Each such token, called a *trailed locality*, is of the form $\omega.l$ where σ is a trail and l is a locality in $\mathcal{L}$ other than `self`. Intuitively, its first part, σ, identifies the invocation in which the token is available, while the second part, l, provides its value. Loc-places labelled by `self` indicate where processes are being executed.
- *Tuple-place:* This is a distinguished place, labelled by $\mathbb{TS}$, used to represent data stored at various tuple spaces. It will store a multiset of structured tokens of the form $l.l'$, each such token corresponding to the expression $l :: \langle l' \rangle$ in the process algebra.

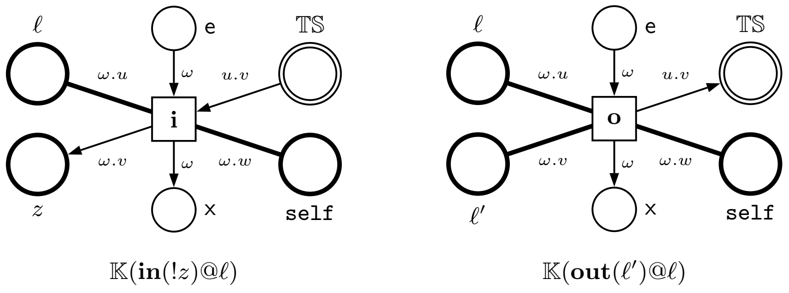


Fig. 2. Translations for two basic actions.

Two example translations for the basic actions are given in figure 2. The first one, $\mathbb{K}(\mathbf{in}(!z)@\ell)$, can match any tuple in the space identified by ℓ. We do not assume that ℓ', ℓ and `self` are distinct, and if that is the case, we collapse the corresponding loc-places, and gather together the annotations of the read arcs. When executed under a binding $\flat$, the translation generates the visible label $\mathbf{i}(\flat(w), \flat(v), \flat(u))$. In the second translation, $\mathbb{K}(\mathbf{out}(\ell')@\ell)$, it may well happen that $\ell = \ell'$ in which case the two loc-places collapse into a single one, and we have two annotations for the only read-arc linking it with the only transition, $\omega.u$ and $\omega.v$. When executed under a binding $\flat$ each of the translations generates the visible label $\mathbf{o}(\flat(w), \flat(v), \flat(u))$. The translation then proceeds in the following four phases (see [5] for details):

Phase I Each process P_i is translated compositionally into $\mathbb{K}(P_i)$ and during this process all non-control places with the same label are being merged.

Phase II For each process definition $A(u_1, \ldots, u_r) \overset{\mathrm{df}}{=} P_A$, we first translate compositionally P_A into $\mathbb{K}(P_A)$ and during this process all non-control places with the same label are being merged into a single one. After that we add loc-place labelled u_i for each $i \leq r$, unless such a place is already present, and suitably deal with the loc-places. The result is denoted $\mathbb{K}(A)$.

Phase III For each network node $l_i :: P_i$, we first translate compositionally P_i into $\mathbb{K}(P_i)$ and during this translation all non-control places with the same label

are being merged. After that, we add loc-place labelled $\texttt{self}_i$ identifying it with the only `self`-labelled place (if present) and give the result label $\texttt{self}_i$.
Phase IV We take the parallel composition of the $\mathbb{K}(A)$'s and $\mathbb{K}(l_i :: P_i)$'s, identifying all non-control places with the same label, and then suitably connect the nets to mimic process instantiation. After that we set the initial marking; in particular, and for each $l'_j :: \langle l''_j \rangle$, we insert a single $l'_j.l''_j$-token into the $\mathbb{TS}$-labelled place.

It can be shown that the labelled transition system of the original process algebraic expression is strongly bisimilar to that of the resulting net, and so the latter can be used for model checking instead of the former.

5 Conclusion

In this paper, we outlined an approach to context aware location-based mobile systems based on CAMA and sketched how to provide it with a formal concurrency semantics in terms of a suitable process algebra. The resulting description can be analysed using efficient model checking techniques in a way which alleviates the state space explosion. The model checking technique adopted in our work is partial order model checking based on Petri net unfoldings, and we briefly described a semantics preserving translation from the process terms used in the modelling of CAMA to a suitable class of high-level Petri nets.

This research was supported by the EC IST grant 511599 (RODIN).

References

1. E.Best, R.Devillers and M.Koutny: *Petri Net Algebra.* EATCS Monographs on TCS, Springer (2001)
2. L. Bettini et al.: *The KLAIM Project: Theory and Practice.* Proc. of Global Computing, Springer, LNCS 2874 (2003) 88–150
3. C.Bryce, C.Razafimahefa and M.Pawlak: *Lana: An Approach to Programming Autonomous Systems.* Proc. of ECOOP'02 (2002) 281–308
4. R.Devillers, H.Klaudel and M.Koutny: *Petri Net Semantics of the Finite π-Calculus.* Proc. of FORTE 2004, Springer, LNCS 3235 (2004) 309–325
5. R.Devillers, H.Klaudel and M.Koutny: *A Petri Net Semantics of a Simple Process Algebra for Mobility.* Technical Report, CS-TR-912, School of Computing Science, University of Newcastle upon Tyne (2005)
6. D.Gelernter: *Generative Communication in Linda.* ACM Computing Surveys 7 (1985) 80–112
7. A.Iliasov, L.Laibinis, A.Romanovsky and E.Troubitsyna: *Towards Formal Development of Mobile Location-Based Systems.* To appear in REFT (2005)
8. V.Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings.* PhD Thesis, School of Computing Science, University of Newcastle upon Tyne (2003)
9. R.Milner, J.Parrow and D.Walker: *A Calculus of Mobile Processes.* Information and Computation 100 (1992) 1–77
10. G.P.Picco, A.L.Murphy, G.-C.Roman: *Lime: Linda Meets Mobility.* Proc. of ICSE'99 (1999)
11. *The Mobile Agent List.* `http://reinsburgstrasse.dyndns.org//mal/preview`

Shortest Violation Traces in Model Checking Based on Petri Net Unfoldings and SAT[*]

Victor Khomenko

School of Computing Science, University of Newcastle
Newcastle upon Tyne NE1 7RU, U.K.
e-mail: `Victor.Khomenko@ncl.ac.uk`

Abstract. Model checking based on the causal partial order semantics of Petri nets is an approach widely applied to cope with the state space explosion problem. One of the possibilities for the verification process is to build a finite and complete prefix and use it for constructing a Boolean formula such that any satisfying assignment to its variables yields a trace violating the property being checked. (And if there are no satisfying assignments then the property holds.)
In this paper a method for computing the *shortest* violation traces (which can greatly facilitate debugging) is proposed. Experimental results demonstrate that it can achieve significant reductions in the size of the Boolean formula as well as in the time required to compute a shortest violation trace, when compared with a naïve approach.
Keywords: Shortest trace, model checking, Petri net unfolding, SAT, Boolean circuit.

1 Introduction and basic notions

A distinctive characteristic of reactive concurrent systems is that their sets of local states have descriptions which are both short and manageable, and the complexity of their behaviour comes from highly complicated interactions with the external environment rather than from complicated data structures and manipulations thereon. One way of coping with this complexity problem is to use formal methods and, especially, computer aided verification tools implementing model checking — a technique in which the verification of a system is carried out using a finite representation of its state space.

The main drawback of model checking is that it suffers from the *state space explosion* problem. That is, even a relatively small system specification can (and often does) yield a very large state space. To cope with this, several techniques have been developed, which usually aim either at a compact representation of the full state space of the system, or at the generation of its reduced (though sufficient for a given verification task) state space. Among them, a prominent technique is McMillan's (finite prefixes of) Petri Net unfoldings (see, e.g., [5, 7]). They rely on the partial order view of concurrent computation, and represent system states implicitly, using an acyclic net, called a *prefix*.

Most of 'interesting' problems for safe Petri nets are $\mathcal{PSPACE}$-complete [2], but the same problems for prefixes are often in $\mathcal{NP}$ or even $\mathcal{P}$. Though the size

[*] The full version of this paper [6] is available on-line.

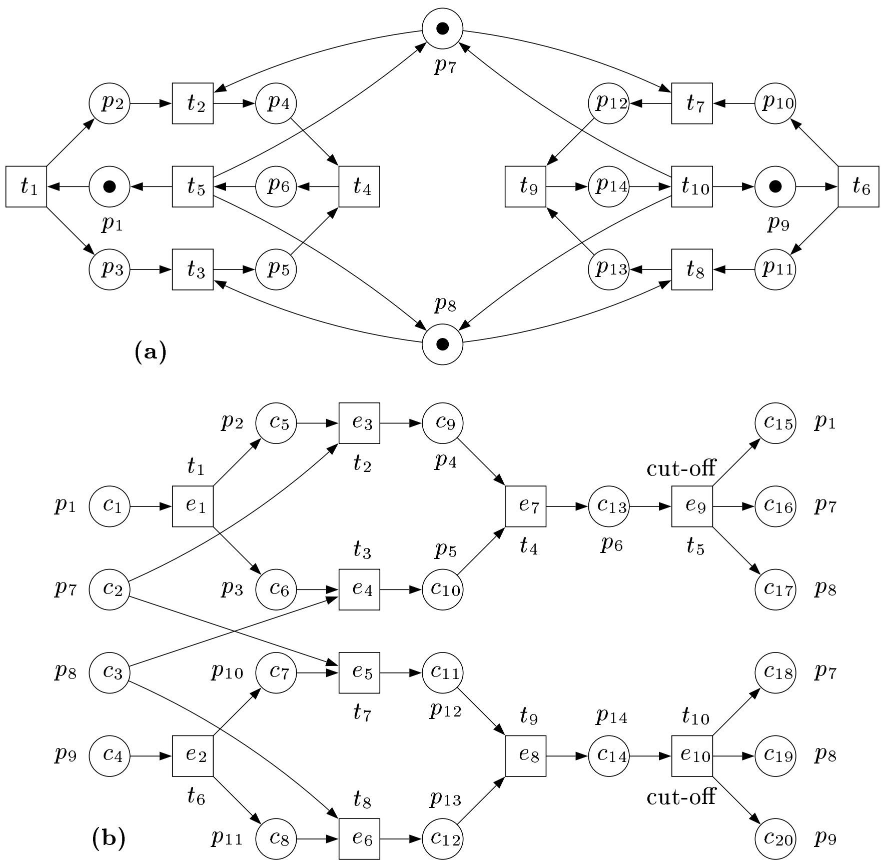
of a finite and complete unfolding prefix can be exponential in the size of the original Petri net, in practice it is often relatively small.

A model checking problem formulated for a prefix can usually be translated into some canonical problem, e.g., Boolean satisfiability (SAT). Then an off-the-shelf SAT solver can be used for efficiently solving it. Such a combination 'unfolder & solver' turns out to be quite powerful in practice.

Petri nets A *net* is a triple $N \overset{\mathrm{df}}{=} (P, T, F)$ such that P and T are disjoint sets of respectively *places* and *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. A *marking* of N is a multiset M of places, i.e., $M : P \to \mathbb{N} \overset{\mathrm{df}}{=} \{0, 1, 2, \ldots\}$. The standard rules about drawing nets are adopted in this paper, viz. places are represented as circles, transitions as boxes, the flow relation by arcs, and the marking is shown by placing tokens within circles. As usual, ${}^\bullet z \overset{\mathrm{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \overset{\mathrm{df}}{=} \{y \mid (z, y) \in F\}$ denote the *pre-* and *postset* of $z \in P \cup T$, and ${}^\bullet Z \overset{\mathrm{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \overset{\mathrm{df}}{=} \bigcup_{z \in Z} z^\bullet$, for all $Z \subseteq P \cup T$. In this paper, the presets of transitions are restricted to be non-empty, i.e., ${}^\bullet t \neq \emptyset$ for every $t \in T$. A *net system* is a pair $\Upsilon \overset{\mathrm{df}}{=} (N, M_0)$ comprising a finite net N and an *initial* marking M_0. It is assumed that the reader is familiar with the standard notions of the Petri nets theory, such as the *enabledness* and *firing* of a transition, marking *reachability* and *deadlock*.

Unfolding prefix A *finite and complete unfolding prefix* π of a Petri net Υ is a finite acyclic net which implicitly represents all the reachable states of Υ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* Υ, by successive firings of transition, under the following assumptions: (a) for each new firing a fresh transition (called an *event*) is generated; (b) for each newly produced token a fresh place (called a *condition*) is generated. The unfolding is infinite whenever Υ has an infinite run; however, if Υ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix. The sets of conditions, events and cut-off events of the prefix are denoted by B, E and E_{cut}, respectively. (Note that $E_{cut} \subseteq E$).

Efficient algorithms exist for building such prefixes [5], which ensure that the number of non-cut-off events $|E \setminus E_{cut}|$ in a complete prefix can never exceed the number of reachable states of Υ. Moreover, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will coincide with the net itself. Another example, viz. a Petri net modelling two dining philosophers, and a finite and complete prefix of its unfolding, are shown in Fig. 1. One can observe that if this example is scaled up, the size of the prefix is linear in the number of dining philosophers, even though the number of reachable states grows exponentially.

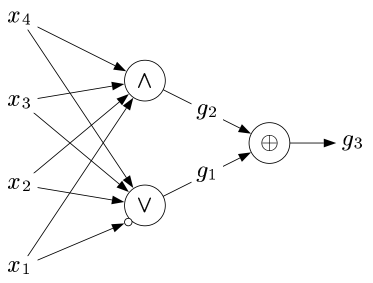


p_7
p_2
t_2
p_4
p_{12}
t_7
p_{10}
t_1
t_5
p_6
t_4
t_9
p_{14}
t_{10}
t_6
p_1
p_9
p_3
t_3
p_5
p_{13}
t_8
p_{11}
p_8
(a)
c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10} c_{11} c_{12} c_{13} c_{14} c_{15} c_{16} c_{17} c_{18} c_{19} c_{20}
e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10}
cut-off
cut-off
(b)

Fig. 1. A Petri net modelling two dining philosophers **(a)** and a finite and complete prefix of its unfolding **(b)**.

Since π is acyclic, the transitive closure of its flow relation is a partial order $<$ on $B \cup E$, called the *causality relation*. (The reflexive order corresponding to $<$ will be denoted by $\leq$.) Intuitively, all the events which are smaller than an event $e \in E$ w.r.t. $<$ must precede e in any valid execution containing e. Two nodes $x, y \in B \cup E$ are in *conflict*, denoted $x\#y$, if there are distinct events $e, f \in E$ such that ${}^\bullet e \cap {}^\bullet f \neq \emptyset$ and $e \leq x$ and $f \leq y$. Intuitively, no valid execution can contain two events in conflict. Two nodes $x, y \in B \cup E$ are *concurrent*, denoted $x \; co \; y$, if neither $y\#y'$ nor $y \leq y'$ nor $y' \leq y$. Intuitively, two concurrent events can be enabled simultaneously, and executed in any order, or even concurrently. For example, in the prefix shown in Fig. 1(b) the following relationships hold: $e_1 < e_7$, $e_7\#e_8$ (due to the choices at c_2 and c_3) and $e_3 \; co \; e_4$.

The reachable markings of Υ can be represented using *configurations* of π. A *configuration* is a set of events $C \subseteq E \setminus E_{cut}$ such that for all $e, f \in C$, $\neg(e\#f)$ and, for every $e \in C$, $f < e$ implies $f \in C$. For example, in the net shown in Fig. 1(b), $\{e_1, e_3, e_4\}$ is a configuration, whereas $\{e_1, e_2, e_3, e_5\}$ and $\{e_1, e_3, e_7\}$



$$\begin{array}{l}
[g_1 \leftrightarrow (\neg x_1 \vee x_2 \vee x_3 \vee x_4)] \wedge \\
[g_2 \leftrightarrow (x_1 \wedge x_2 \wedge x_3 \wedge x_4)] \wedge \\
[g_3 \leftrightarrow (g_1 \oplus g_2)] \\
\qquad\qquad \equiv \\
[(\neg g_1 \vee \neg x_1 \vee x_2 \vee x_3 \vee x_4) \wedge \\
(g_1 \vee x_1) \wedge (g_1 \vee \neg x_2) \wedge \\
\ (g_1 \vee \neg x_3) \wedge (g_1 \vee \neg x_4)] \wedge \\
[(g_2 \vee \neg x_1 \vee \neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge \\
(\neg g_2 \vee x_1) \wedge (\neg g_2 \vee x_2) \wedge \\
\ (\neg g_2 \vee x_3) \wedge (\neg g_2 \vee x_4)] \wedge \\
[(\neg g_1 \vee \neg g_2 \vee \neg g_3) \wedge (\neg g_1 \vee g_2 \vee g_3) \wedge \\
\ (g_1 \vee \neg g_2 \vee g_3) \wedge (g_1 \vee g_2 \vee \neg g_3)]
\end{array}$$

Fig. 2. Conversion of a Boolean circuit into a Boolean expression in the CNF.

are not (the former includes events in conflict, $e_3\#e_5$, while the latter does not include $e_4 < e_7$). Intuitively, a configuration is a partial-order execution, i.e., an execution where the order of firing of some of its events (viz. concurrent ones) is not important; e.g., the configuration $\{e_1, e_3, e_4, e_7\}$ corresponds to two totally ordered executions: $e_1e_3e_4e_7$ and $e_1e_4e_3e_7$. Since a configuration can correspond to multiple executions, it is often much more efficient in model checking to explore configurations rather than executions.

After starting π from the implicit initial marking (whereby one puts a single token in each condition which does not have an incoming arc) and executing all the events in C, one reaches the marking denoted by $\mathit{Cut}(C)$. $\mathit{Mark}(C)$ denotes the corresponding marking of Υ, reached by firing a transition sequence corresponding to the events in C. It is remarkable that each reachable marking of Υ is $\mathit{Mark}(C)$ for some configuration C of π, and, conversely, each configuration C of π generates a reachable marking $\mathit{Mark}(C)$. Thus various behavioural properties of Υ can be re-stated as the corresponding properties of π, and then checked, often much more efficiently.

Boolean satisfiability The *Boolean satisfiability problem (SAT)* consists in finding a *satisfying assignment*, i.e., a mapping $A : \mathit{Var}_\varphi \to \{0,1\}$ defined on the set of variables Var_φ occurring in a given Boolean expression φ such that φ evaluates to 1. This expression is often assumed to be given in the *conjunctive normal form (CNF)* $\varphi = \bigwedge_{i=1}^{n} \bigvee_{l \in L_i} l$, i.e., it is represented as a conjunction of *clauses*, which are disjunctions of *literals*, each literal l being either a variable or the negation of a variable. It is assumed that no two literals in the same clause correspond to the same variable.

In order to solve a Boolean satisfiability problem, SAT solvers perform exhaustive search assigning the values 0 or 1 to the variables, using heuristics to reduce the search space [10]. Some of the leading SAT solvers, e.g., zChaff [8], can be used in the *incremental mode,* i.e., after solving a particular SAT instance the user can slightly change it (e.g., by adding and/or removing a small number of clauses) and execute the solver again. This is often much more efficient than solving these related instances as independent problems, because on the subsequent runs the solver can use some of the useful information (e.g., learnt clauses [10]) collected so far.

Boolean circuits A *Boolean circuit* (see, e.g., [9]) computes a multiple-output Boolean function of Boolean *input variables* $x_1, \ldots, x_n$. It consists of a finite number k of *gates* $G_1, \ldots, G_k$. Each gate G_i is labelled by a Boolean function f_i chosen from some fixed set of Boolean functions $\mathcal{F}$. (In this paper, $\mathcal{F}$ comprises all the unary and binary Boolean functions and conjunctions and disjunctions of arbitrary arity with arbitrary input inversions.) A Boolean circuit can be represented by an acyclic directed graph, where the input variables and the constants 0 and 1 are its sources, and the vertex representing the gate G_i has $\mathit{arity}(f_i)$ numbered incoming edges from its predecessors in the graph. (If f_i is commutative, the numbering of edges does not have to be specified.) In pictures, each gate is represented as a circle with the function shown within it, and input inversions are shown as 'bubbles'. Note that $\mathcal{F}$ is closed w.r.t. input inversions, and so they can be incorporated into the corresponding gate function.

The Boolean function f_v computed at a vertex v of this acyclic graph is defined inductively as follows. If v is an input variable x_j then $f_v(x_1, \ldots, x_n) \overset{\mathrm{df}}{=} x_j$, and if it is a constant $c \in \{0, 1\}$ then $f_v(x_1, \ldots, x_n) \overset{\mathrm{df}}{=} c$. Otherwise, the vertex is some gate G_i, and $f_v(x_1, \ldots, x_n) \overset{\mathrm{df}}{=} f_i(p_1, \ldots, p_{\mathit{arity}(f_i)})$, where $p_1, \ldots, p_{\mathit{arity}(f_i)}$ are the functions computed at the predecessors of this vertex in the graph. The *output vector* $(v_1, \ldots, v_m)$, where v_i is some vertex of the graph, describes what the circuit computes, viz. the multiple-output Boolean function $(f_{v_1}, \ldots, f_{v_m})$. In particular, any Boolean formula over the signature $\mathcal{F}$ can be represented as a circuit.

It turns out that a Boolean circuit can be efficiently encoded by a Boolean expression φ in the CNF depending on the variables Var_φ corresponding to the vertices of the graph representing the circuit (except 0 and 1) such that for any assignment $A : \mathit{Var}_\varphi \to \{0, 1\}$, A is a satisfying assignment of φ iff for every $v \in \mathit{Var}_\varphi$, $f_v(A(x_1), \ldots, A(x_n)) = A(v)$ (where the variables are denoted by the same symbol as the corresponding vertices of the graph) and $A(0) \overset{\mathrm{df}}{=} 0$ and $A(1) \overset{\mathrm{df}}{=} 1$.

The expression φ is constructed as follows. For each gate G_i, a new Boolean variable g_i representing its output is created, a Boolean equation relating g_i to the inputs of G_i is written down, and these equations are converted into the CNF. This process is illustrated in Fig. 2. Note that for a gate labelled with a Boolean function of bounded arity, the size of the corresponding equation (and its CNF) is bounded by a constant; moreover, for a gate labelled with a multiple-input conjunction or disjunction, the size of the equation (and its CNF) is linear in the number of gate inputs. Thus the size of the resulting Boolean expression in the CNF is linear in the size of the circuit.

Model checking based on Petri net unfoldings This paper concentrates on the following approach to model checking. First, a finite and complete prefix of the Petri net unfolding is built, and it is then used for constructing a Boolean formula encoding the model checking problem at hand. (It is assumed that the property being checked is the unreachability of some 'bad' states, e.g., deadlocks.) This formula is unsatisfiable iff the property holds, and such that any satisfying assignment to its variables yields a trace violating the property being checked.

Typically such a formula would have for each non-cut-off event e of the prefix a variable conf_e (the formula might also contain other variables), and for every satisfying assignment A, the set of events $C \overset{\mathrm{df}}{=} \{e \mid \mathsf{conf}_e = 1\}$ is a configuration such that $\mathit{Mark}(C)$ violates the property being checked. The formula often has the form $\mathcal{CONF} \wedge \mathcal{VIOL}$. The role of the *configuration constraint, $\mathcal{CONF}$*, is to ensure that C is a configuration of the prefix (not just an arbitrary set of events). $\mathcal{CONF}$ can be defined as the conjunction of the formulae

$$\bigwedge_{e\in E\setminus E_{cut}} \bigwedge_{f\in{}^\bullet({}^\bullet e)} (\mathsf{conf}_e \to \mathsf{conf}_f) \quad \text{and} \quad \bigwedge_{e\in E\setminus E_{cut}} \bigwedge_{f\in(({}^\bullet e)^\bullet\setminus\{e\})\setminus E_{cut}} \neg(\mathsf{conf}_e \wedge \mathsf{conf}_f)\ .$$

The former formula ensures that if $e \in C$ then its immediate predecessors are also in C, i.e., C is downward closed w.r.t. $<$. The latter one ensures that C contains no conflicts. $\mathcal{CONF}$ can be transformed into the CNF by applying the rules $x \to y \equiv \neg x \vee y$ and $\neg(x \wedge y) \equiv \neg x \vee \neg y$. For example, the configuration constraint for the prefix shown in Fig. 1(b) is

$$\begin{aligned}&(\mathsf{conf}_{e_3}\to\mathsf{conf}_{e_1})\wedge(\mathsf{conf}_{e_4}\to\mathsf{conf}_{e_1})\wedge(\mathsf{conf}_{e_5}\to\mathsf{conf}_{e_2})\wedge\\ &\quad(\mathsf{conf}_{e_6}\to\mathsf{conf}_{e_2})\wedge(\mathsf{conf}_{e_7}\to\mathsf{conf}_{e_3})\wedge(\mathsf{conf}_{e_7}\to\mathsf{conf}_{e_4})\wedge\\ &\quad(\mathsf{conf}_{e_8}\to\mathsf{conf}_{e_5})\wedge(\mathsf{conf}_{e_8}\to\mathsf{conf}_{e_6})\wedge\neg(\mathsf{conf}_{e_3}\wedge\mathsf{conf}_{e_5})\wedge\neg(\mathsf{conf}_{e_4}\wedge\mathsf{conf}_{e_6})\ .\end{aligned}$$

The role of the *violation constraint, $\mathcal{VIOL}$*, is to express the property violation condition for a configuration C, so that if a configuration C satisfying this constraint is found then the property does not hold, and any ordering of events in C consistent with $<$ is a violation trace. For example, for deadlock checking $\mathcal{VIOL}$ can be defined as

$$\bigwedge_{e\in E}\Big(\bigvee_{f\in{}^\bullet({}^\bullet e)}\neg\mathsf{conf}_f \vee \bigvee_{f\in({}^\bullet e)^\bullet\setminus E_{cut}}\mathsf{conf}_f\Big)\ .$$

This formula requires for each event e (including cut-off events) that some of the direct causal predecessors of e has not fired or some of the non-cut-off events (including e unless it is cut-off) consuming tokens from ${}^\bullet e$ has fired, and thus e is not enabled. This formula is already in the CNF. For example, the violation constraint for the deadlock checking problem formulated for the prefix shown in Fig. 1(b) is

$$\begin{aligned}&\mathsf{conf}_{e_1}\wedge\mathsf{conf}_{e_2}\wedge(\neg\mathsf{conf}_{e_1}\vee\mathsf{conf}_{e_3})\wedge(\neg\mathsf{conf}_{e_1}\vee\mathsf{conf}_{e_4})\wedge\\ &\quad(\neg\mathsf{conf}_{e_2}\vee\mathsf{conf}_{e_5})\wedge(\neg\mathsf{conf}_{e_2}\vee\mathsf{conf}_{e_6})\wedge(\neg\mathsf{conf}_{e_3}\vee\neg\mathsf{conf}_{e_4}\vee\mathsf{conf}_{e_7})\wedge\\ &\quad(\neg\mathsf{conf}_{e_5}\vee\neg\mathsf{conf}_{e_6}\vee\mathsf{conf}_{e_8})\wedge\neg\mathsf{conf}_{e_7}\wedge\neg\mathsf{conf}_{e_8}\ .\end{aligned}$$

Shortest violation traces Note that in general the computed violation trace can be quite long, which might make it difficult to locate the error, as the designer has to inspect this trace in order to find and eliminate the source of the problem. (And parts of such long traces often describe incidental system activity which is unrelated to the problem.) Thus computing shortest possible violation traces can greatly facilitate the debugging process.

A quite obvious algorithm for computing the shortest violation trace is shown in Fig. 3, where $\mathit{SAT\_Assignment}(\varphi)$ is a function computing a satisfying assignment for a Boolean formula φ and returning UNSAT in case φ is unsatisfiable (it is usually implemented by a call to some off-the-shelf SAT solver,

```
input : $\varphi$ — a Boolean formula
output : $T$ — the shortest violation trace or UNSAT

$A \leftarrow SAT\_Assignment(\varphi)$
if $A =$ UNSAT
    then
        $T \leftarrow$ UNSAT
        stop
$T \leftarrow Extract\_Trace(A)$
$r \leftarrow |T|$
$l \leftarrow 0$
while $l < r$ do
    $t \leftarrow \lceil (l + r)/2 \rceil$
    $A \leftarrow SAT\_Assignment(\varphi \wedge Threshold_t)$
    if $A =$ UNSAT
    then
        $l = t + 1$
    else
        $T \leftarrow Extract\_Trace(A)$
        $r \leftarrow |T|$
```

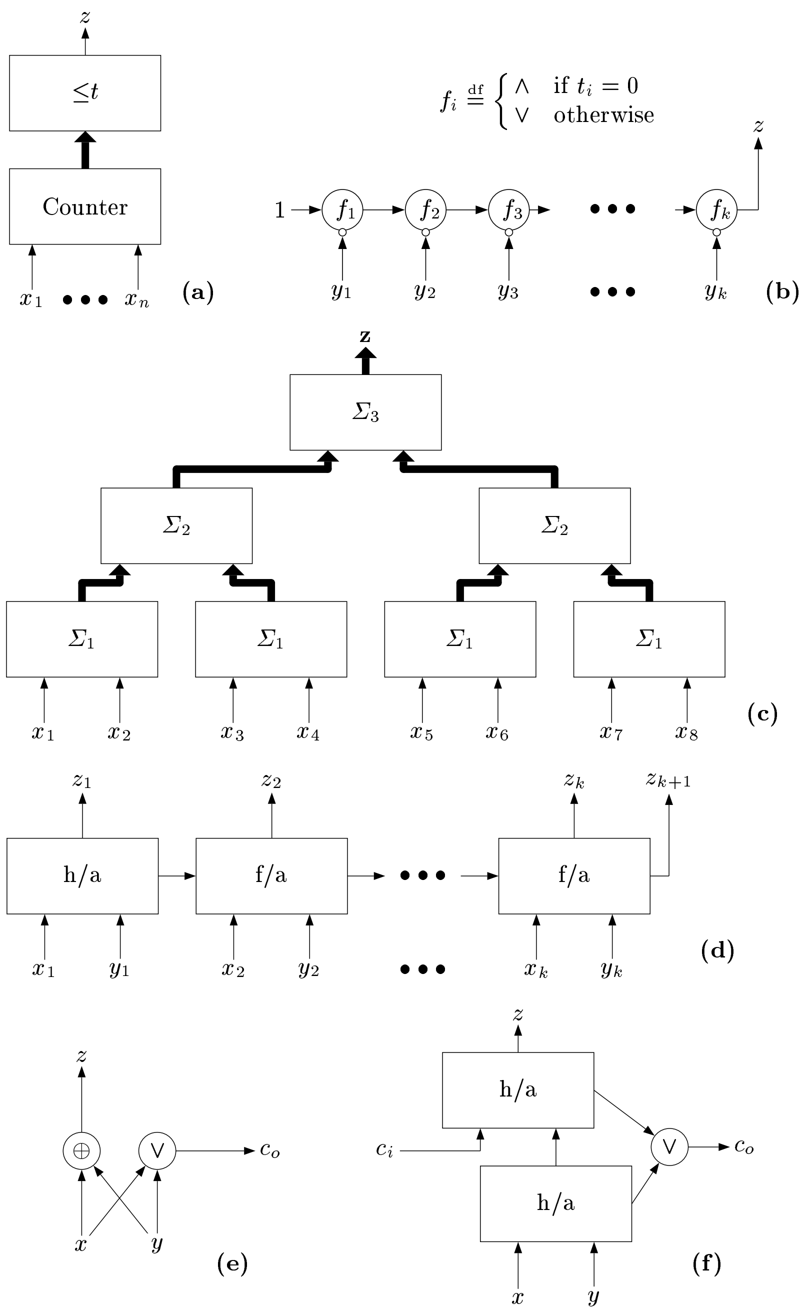
Fig. 3. An algorithm for computing shortest violation traces.

e.g., zChaff [8]), *Extract_Trace*(A) is a function extracting the violation trace from a satisfying Boolean assignment A, and $Threshold_t$ is the *threshold constraint* $|\{e \mid \mathsf{conf}_e = 1\}| \leq t$. This algorithm uses a binary search to compute the length of the shortest trace still exhibiting the violation. If the property holds (i.e., if φ is unsatisfiable) then this algorithm does not have any additional overhead compared with the original model checking algorithm, but in the case of errors the SAT solver is called several times with larger formulae, and so the overhead might be quite significant. This situation is somewhat alleviated by the fact that SAT instances are very similar to each other (in fact, even the formulae of the form $Threshold_t$, described in detail further in this paper, change very little when t changes) and thus can be efficiently solved in the incremental mode. Moreover, the user always can terminate the execution of the algorithm and get the shortest violation trace computed so far.

What still needs describing is the construction of the formula $Threshold_t$ for a given t. It turns out that one can exploit some problem-specific optimisations in order to significantly reduce the size of this formula as well as the computation effort required for solving the corresponding SAT instances. This is the main topic of this paper.

2 Basic translation of a threshold constraint

$Threshold_t$ can be expressed as a *pseudo-Boolean* constraint $\sum_{e \in E \setminus E_{cut}} \mathsf{conf}_e \leq t$, where arithmetical operations are used instead of logical ones. The other constraints can also be converted into a similar form, and the problem can be solved by a 0–1 integer linear programming solver. However, SAT solvers tend to be



z
$\leq t$
Counter
x_1 ••• x_n
(a)
$f_i \overset{\mathrm{df}}{=} \begin{cases} \wedge & \text{if } t_i = 0 \\ \vee & \text{otherwise} \end{cases}$
1
f_1 f_2 f_3 ••• f_k
z
y_1 y_2 y_3 ••• y_k
(b)
z
Σ_3
Σ_2 Σ_2
Σ_1 Σ_1 Σ_1 Σ_1
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8
(c)
z_1 z_2 z_k z_{k+1}
h/a f/a ••• f/a
x_1 y_1 x_2 y_2 ••• x_k y_k
(d)
z
$\oplus$ $\vee$
c_o
x y
(e)
z
h/a
c_i
$\vee$
c_o
h/a
x y
(f)

Fig. 4. Implementations of a threshold constraint (**a**); a comparator (**b**), where the inputs $y_1, \ldots, y_k$ are interpreted as the binary representation of a non-negative integer (least significant digit first) and $t_1, \ldots, t_k$ is the binary representation of t; a counter as a balanced tree of adders (**c**); a k-bit adder Σ_k comprising a half-adder cell and $k-1$ full-adder cells (**d**); and half-adder and full-adder cells (**e,f**).

more efficient in practice, and so in many cases it would be advantageous to express $\mathit{Threshold}_t$ as a purely Boolean constraint.

A possible implementation of $\mathit{Threshold}_t$ as a Boolean circuit is shown in Fig. 4(a). It consists of two parts: the counter and the comparator. The counter circuit has n inputs and $\lceil \log_2 n \rceil + 1$ outputs, and its purpose is to count the number of ones among its inputs and return the result as a binary number. The purpose of the comparator is to compare this number with a given constant t.

Note that the counter circuit does not depend on t and so the corresponding part of the formula does not have to be changed between the calls to the SAT solver in the algorithm shown in Fig. 3. A possible implementation of the comparator is shown in Fig. 4(b). Note that it does depend on t, and so the corresponding part of the formula has to be amended from call to call. However, the size of the comparator is just $O(\log n)$. Thus this implementation of the threshold constraint is beneficial if the SAT solver is used in the incremental mode. The rest of this section is devoted to the counter circuit.

Fig. 4(c) illustrates an implementation of the counter as a tree of adders, where each adder is built of half-adder and full-adder cells, as shown in Fig. 4(d). A half-adder cell adds up two one-bit numbers, producing a one-bit result and a carry bit. A full-adder cell adds up two one-bit numbers and a carry from the previous cell of the adder, producing a one-bit result and a carry bit. Fig. 4(e,f) shows possible implementations of these cells.

The described circuit can be converted to a linear-size formula in the CNF, as described in Section 1. However, somewhat shorter formulae can be obtained using Boolean minimisation when translating half-adder and full-adder cells. It yields the formulae

$$(\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z) \wedge (y \vee \neg c_o) \wedge (\neg x \vee c_o \vee z) \wedge (\neg c_o \vee \neg z)$$

with 2 new variables, 6 clauses and 16 literals for a half-adder cell, and

$$(c_i \vee \neg x \vee y \vee z) \wedge (c_i \vee x \vee \neg y \vee z) \wedge (\neg c_i \vee \neg x \vee y \vee \neg z) \wedge (\neg c_i \vee x \vee \neg y \vee \neg z) \wedge (\neg c_i \vee c_o \vee z) \wedge$$
$$(c_i \vee \neg c_o \vee \neg z) \wedge (\neg x \vee \neg y \vee c_o) \wedge (x \vee y \vee \neg c_o) \wedge (\neg c_i \vee \neg x \vee \neg y \vee z) \wedge (c_i \vee x \vee y \vee \neg z)$$

with 2 new variables, 10 clauses and 36 literals for a full-adder cell.

It is shown in [6] that if n is a power of 2 then the resulting CNF formula for the counter contains $4n - 2\log_2 n - 4$ auxiliary variables (corresponding to gate outputs), $16n - 10\log_2 n - 16$ clauses and $52n - 36\log_2 n - 52$ literals, i.e., even though the size of the formula is linear in the number of the circuit's inputs, the multiplicative constants hidden in this $O(n)$ translation are quite large. Next section tries to remedy this situation by exploiting the structure of the prefix to improve the described translation.

3 Exploiting the structure of the prefix

The content of this section is the main contribution of this paper. It turns out that the structure of the prefix can be exploited to reduce the size of the counter circuit. Below, two heuristics are described, one utilising the conflicts between the events in the prefix, and the other making use of the causality relation.

Exploiting the conflicts One can observe that if $E' \subseteq E \setminus E_{cut}$ is a set of events which are in conflict with each other (i.e., E' is a clique in the graph corresponding to the relation $\#$) then no two events from E' can belong to the same configuration. The configuration constraint ensures that at most one of the variables conf_e corresponding to the events in E' is assigned the value 1, i.e., $1 \geq |\{e \in E' \mid \mathsf{conf}_e = 1\}| = \bigvee_{e \in E'} \mathsf{conf}_e$, and so a single $\vee$-gate is sufficient to count the number of variables assigned the value 1.

Definition 1 (#-cluster). *A set of events $E' \subseteq E \setminus E_{cut}$ is a* $\#$-cluster *if for all distinct events $e, f \in E'$, $e \# f$.*

Thus the non-cut-off events of the prefix are partitioned into $\#$-clusters, then $\vee$-gates are used to count in each $\#$-cluster the number of variables corresponding to its events and assigned the value 1, and a counter (hopefully, of a much smaller size) is used to count the number of outputs of these $\vee$-gates having the value 1. Since the translation of an $\vee$-gate into a Boolean expression is much smaller than the translation of a counter, one can expect reductions in the size of the resulting formula. For example, $\{\{e_1\}, \{e_2\}, \{e_3, e_5\}, \{e_4, e_6\}, \{e_7, e_8\}\}$ is a possible partition into $\#$-clusters of the non-cut-off events of the prefix shown in Fig. 1(b).

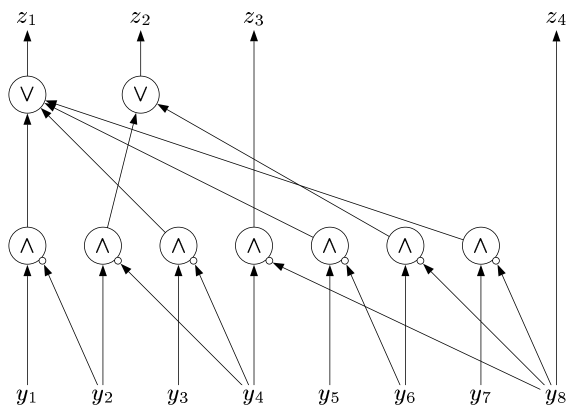
When partitioning the non-cut-off events of the prefix into $\#$-clusters, it is advantageous to make the number of such $\#$-clusters as small as possible. (When the number of $\#$-clusters is large, the size of the counter grows; in particular, for the trivial partition with each event forming its own $\#$-cluster the translation degrades to the one described in the previous section.) Thus one can formulate an optimisation problem of partitioning the non-cut-off events of a prefix into the smallest number of $\#$-clusters. Unfortunately, a decision version of this problem turns out to be NP-complete.

Proposition 1 (NP-completeness of the Partition into #-clusters problem). *Given an unfolding prefix π and a $k \in \mathbb{N}$, the problem of deciding whether the set of non-cut-off events of π can be partitioned into at most k $\#$-clusters is NP-complete.*

The proof is by reduction from the *Partition into Cliques* problem, which is known to be NP-complete [3, Problem GT15], and can be found in [6].

When computing the shortest violation trace, one does not want to spend too much effort on building the threshold constraints, as the process of building them can easily become more time consuming then model checking itself. Therefore, in the actual implementation, a fast 'greedy' algorithm for partitioning the set of events into $\#$-clusters was adopted, which is justifiable in the view of the above result. This algorithm is described in [6].

Exploiting the causality relation The method described above allowed for simplification of the threshold constraint by exploiting the conflict relation between the events in the prefix. It turns out that the causality relation can also be exploited to reduce the size of the translation even further.



$$z_i \overset{\mathrm{df}}{=} \bigvee_{j=0}^{\left\lceil \frac{k-2^{i-1}}{2^i} \right\rceil} y_{2^i(j+\frac{1}{2})} \wedge \neg y_{2^i(j+1)}$$
$$i \in \{1, \ldots, \lceil \log_2 k \rceil + 1\}$$
$$y_{k'} \overset{\mathrm{df}}{=} 0 \text{ if } k' > k$$

Fig. 5. An implementation of an eight-input counter with the values of inputs constrained to be in a non-increasing order.

Definition 2. *Let Cl and Cl′ be two #-clusters. $Cl \ll Cl'$ if for each event $e' \in Cl'$ there exists an event $e \in Cl$ such that $e < e'$. A sequence of #-clusters $Cl_1 \ll Cl_2 \ll \cdots \ll Cl_k$ is called a* $\ll$-chain.

For example, $\{e_4, e_6\} \ll \{e_7, e_8\}$ is a $\ll$-chain of the prefix shown in Fig. 1(b).

It follows from this definition that if $Cl \ll Cl'$ and an event $e' \in Cl'$ belongs to a configuration C then some event $e \in Cl$ also belongs to C. Suppose $Cl_1 \ll Cl_2 \ll \cdots \ll Cl_k$ is a $\ll$-chain and $y_1, \ldots, y_k$ are the outputs of the $\vee$-gates corresponding to these #-clusters. The configuration constraint ensures that in any satisfying assignment *the sequence of values of $y_1, \ldots, y_k$ is non-increasing.* This allows one to count the number of ones among these values much more efficiently than by a counter described in the previous section. Indeed, the encoding of the inputs is very similar to the 1-hot encoding, which can be obtained from $y_1, \ldots, y_k$ as $\neg y_1, y_1 \wedge \neg y_2, y_2 \wedge \neg y_3, \ldots, y_{k-1} \wedge \neg y_k, y_k$ and subsequently converted into the binary code using an encoder. A somewhat smaller circuit is shown in Fig. 5.

Thus one can partition the acyclic directed graph $G_{\ll}$ corresponding to the $\ll$ relation on the #-clusters into $\ll$-chains, then build for each $\ll$-chain a circuit similar to the one shown in Fig. 5, and finally construct an adder tree similar to that in Fig. 4(c), but with the bottom layer comprised of the built counters rather than half-adders. The algorithm shown in Fig. 6 does this trying to balance the resulting adder tree. *ExtractMin*(Q) extracts and returns a pair $(c, m) \in Q$ (where c is a circuit and $m \in \mathbb{N}$ is the maximum value this circuit can output) with the minimum value of m, and *Add*(c_1, c_2) constructs a circuit which computes the sum of values computed by c_1 and c_2 (i.e., an adder is put 'on top' of c_1 and c_2). Note that Q is a priority queue and can be efficiently implemented as either a binary heap or by keeping a list of circuits for each m.

When partitioning $G_{\ll}$ into $\ll$-chains, it is advantageous to make the number of such $\ll$-chains as small as possible, in order to reduce the number of adders in the adder tree. Thus one can formulate an optimisation problem of partitioning

```
**input** : $Q$ — a non-empty set of pairs $(c, m)$, where $c$ is a circuit and $m \in \mathbb{N}$
**output** : $c$ — a circuit

**while** $|Q| > 1$ **do**
    $(c_1, m_1) \leftarrow \mathit{ExtractMin}(Q)$
    $(c_2, m_2) \leftarrow \mathit{ExtractMin}(Q)$
    $Q \leftarrow Q \cup \{(\mathit{Add}(c_1, c_2), m_1 + m_2)\}$

/* now $|Q|$=1 */
$(c, m) \leftarrow \mathit{ExtractMin}(Q)$
**return** c
```

Fig. 6. An algorithm for building a tree of adders.

$G_{\ll}$ into the smallest number of $\ll$-chains. This is essentially the well-known *minimum vertex-disjoint path cover* problem (zero-length paths comprising a single vertex are admissible).

This problem is NP-complete for general graphs, since checking the existence of a Hamiltonian path is equivalent to checking whether it is possible to cover the vertices of a given graph by a single vertex-disjoint path. Nevertheless, for acyclic graphs (note that $G_{\ll}$ is acyclic) it can be reduced to the maximum matching problem on a bipartite graph, and solved in polynomial time [4]. However, one should bear in mind that $G_{\ll}$ is given implicitly, and can be very large. (It is not uncommon to have an unfolding prefix with hundreds thousands events.) Therefore, using an exact algorithm for solving this problem might be either too memory demanding (if $G_{\ll}$ is built explicitly), or too slow due to the need of working with an implicitly represented graph (checking whether there is an arc between two vertices of $G_{\ll}$ is quite expensive in such a case, as one might have to traverse the whole prefix). Thus a fast 'greedy' algorithm for partitioning the set of #-clusters into $\ll$-chains has been designed. It is described in [6].

4 Experimental results

The proposed method has been tested with the zChaff SAT solver [8], and the popular set of deadlock checking benchmarks collected by J.C. Corbett [1] has been attempted. (For obvious reasons, only examples with deadlocks from this collection were used.) All the experiments were conducted on a PC with a Pentium^{TM} IV/2.8GHz processor and 512M RAM.

The experimental results are shown in Table 1, where the meaning of the columns is as follows (from left to right): the name of the problem; the number of non-cut-off events in the prefix; the lengths of the first computed and a shortest violation traces; the number of #-clusters and $\ll$-chains computed by the heuristic algorithms described in [6]; the size (the number of new variables, clauses and literals) of the translation of the counter circuit for the basic translation described in Section 2 and for the improved one described in Section 3; and the time taken by the SAT solver to compute the first violation trace and the time taken by the algorithm in Fig. 3 to compute a shortest violation trace using the basic and the improved translations of the counter.

Problem	Prefix	Trace		Partitions		Translation of counter						Time		
						Basic			Improved					
	$\lvert E\setminus E_{cut}\rvert$	1st	shtst	#-cl	$\ll$-ch	vars	cls	lits	vars	cls	lits	1st	Bas.	Imp.
Q	7229	75	21	179	25	28881	115479	375221	520	8781	26031	<1	3	1
SPEED	1663	24	4	30	9	6620	26436	85832	98	1952	5806	<1	1	<1
DAC(6)	53	6	6	23	11	195	761	2437	72	279	833	<1	<1	<1
DAC(9)	95	9	9	35	17	359	1409	4527	116	460	1372	<1	<1	<1
DAC(12)	146	12	12	47	23	564	2236	7230	160	662	2000	<1	<1	<1
DAC(15)	206	43	15	59	29	802	3182	10292	205	864	2600	<1	<1	<1
DP(6)	66	6	6	18	6	247	973	3135	55	222	628	<1	<1	<1
DP(8)	120	8	8	24	8	461	1823	5885	75	341	987	<1	<1	<1
DP(10)	190	10	10	30	10	737	2919	9431	96	475	1381	<1	<1	<1
DP(12)	276	12	12	36	12	1082	4306	13954	119	635	1861	<1	<1	<1
ELEV(1)	98	9	9	16	5	374	1478	4770	43	222	640	<1	<1	<1
ELEV(2)	496	22	12	24	7	1960	7812	25336	65	685	2017	<1	<1	<1
ELEV(3)	2266	30	15	32	9	9033	36095	117239	94	2549	7607	<1	<1	<1
ELEV(4)	9598	23	18	40	11	38354	153366	498344	117	9950	29798	2	27	3
HART(25)	101	26	26	76	26	385	1519	4897	218	826	2528	<1	<1	<1
HART(50)	201	51	51	151	51	783	3109	10061	440	1684	5188	<1	<1	<1
HART(75)	301	76	76	226	76	1180	4692	15196	666	2566	7942	<1	<1	<1
HART(100)	401	101	101	301	101	1581	6299	20425	888	3424	10602	<1	<1	<1
KEY(2)	454	52	42	103	18	1792	7140	23152	285	1309	3761	<1	<1	<1
KEY(3)	4057	53	43	223	41	16194	64730	210284	680	6123	18051	<1	20	2
KEY(4)	35905	65	44	407	82	143582	574286	1866352	1269	39797	118855	<1	548	224
MMGT(1)	38	6	6	11	2	136	528	1686	25	98	250	<1	<1	<1
MMGT(2)	385	8	8	26	7	1518	6050	19622	80	618	1806	<1	<1	<1
MMGT(3)	3312	10	10	36	6	13217	52831	171631	98	3584	10658	<1	<1	<1
MMGT(4)	25945	12	12	44	7	103741	414915	1348381	119	26273	78693	77	86	80
SENT(25)	176	34	3	40	3	684	2716	8790	69	370	1028	<1	<1	<1
SENT(50)	201	59	3	65	3	783	3109	10061	98	480	1302	<1	<1	<1
SENT(75)	226	84	3	90	3	883	3509	11361	123	579	1549	<1	<1	<1
SENT(100)	251	109	3	115	3	980	3888	12574	149	681	1803	<1	<1	<1

Table 1. Experimental results for deadlock checking.

The experiments show that in many cases the first computed violation trace was much longer than a shortest one, with the results for the SENT benchmarks being particularly impressive. This confirms that in practice computing shortest violation traces can indeed greatly facilitate the debugging process.

One can see that the number of #-clusters and $\ll$-chains is usually quite small compared to the number of non-cut-off events in the prefix, and thus the reduction in the size of the formula is quite significant. It is possible to evaluate the maximum reduction which can be achieved by the improved translation over the basic one as follows. In the ideal case, all the events in the prefix would be in conflict with each other, and so the counter circuit can be implemented as a single $\vee$-gate. Such an implementation results in one new variable (for the gate's output), $n+1$ clauses and $3n+1$ literals in the corresponding CNF formula, where $n = |E \setminus E_{cut}|$. The corresponding parameters for the basic translation are given in Section 2, and the improvement ratios for new variables, clauses and literals are $(4n - 2\log_2 n - 4)/1 \approx 4n$, $(16n - 10\log_2 n - 16)/(n+1) \approx 16$ and $(52n - 36\log_2 n - 52)/(3n+1) \approx 17\frac{1}{3}$, respectively. Thus the reduction ratio for variables can grow unboundedly with n, whereas for clauses and literals it is bounded by 16 and $17\frac{1}{3}$, respectively.

The improvement ratios for the benchmarks in Table 1 are plotted in Fig. 7. One can see that for the number of new variables, the reduction ratio indeed grows with the size of the prefix (though not as fast as in the ideal case), and is

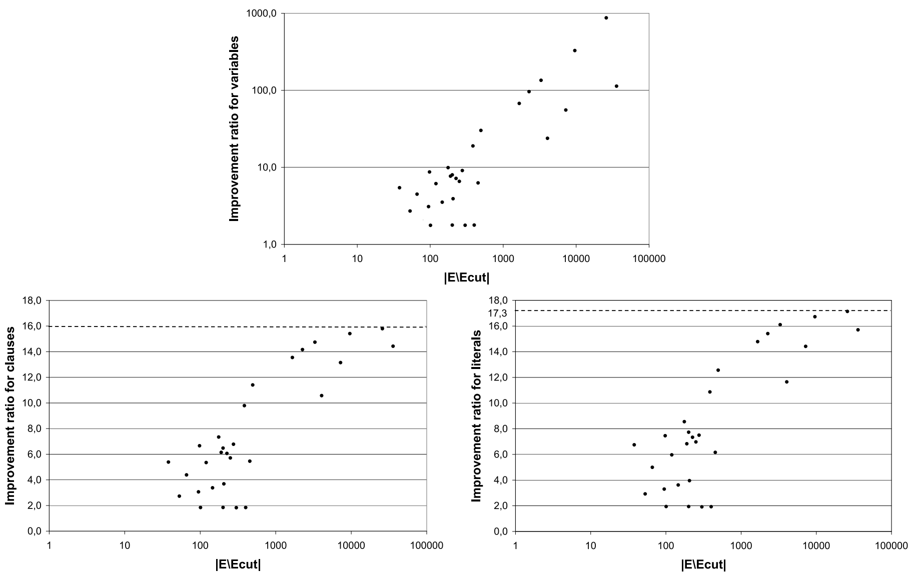


Fig. 7. Improvement ratios.

between two and three orders of magnitude for large benchmarks. For clauses and literals, the improvement rate also grows with the size of the prefix, and comes surprisingly close to the best possible ratio for large benchmarks. Moreover, it should be noted that since the improved translation uses a lot of multiple-input $\vee$-gates, the corresponding CNF formula has many clauses of length two, which makes the SAT instance easier for the solver.

The comparison of the running times of the algorithms shows that, except one test case, it was not too time-consuming to compute a shortest violation trace. (This is probably due to the fact that only a few benchmarks are large.) Moreover, the improved approach has a clear advantage over the basic one in terms of time. The only benchmark where computing the shortest violation trace by the improved method took significantly more time than just solving the original model checking problem was Key(4). (Note that for Mmgt(4) the increase in time was quite modest, which can be explained by the fact that the first computed violation trace was already optimal and very short.) In general, however, one can expect a significant increase in time when computing the shortest violation traces, due to the following phenomenon, related to *phase transition*. Let t^* be the length of the shortest violation trace. If t is significantly larger than t^*, adding the constraint $\mathit{Threshold}_t$ to the formula will exclude only a few satisfying assignments, and the resulting formula will not be much harder for the solver than the original one. On the other hand, if t is significantly smaller than t^*, adding $\mathit{Threshold}_t$ to the formula will yield an overconstrained SAT instance which usually can be quickly proven unsatisfiable. A hard situation can occur when t is close to t^*. In such a case, if the SAT instance is satisfiable, it often has only a small number of satisfying assignments (and thus such an assignment might be difficult to find), and if it is unsatisfiable, it might be hard to show this. The last part of Section 1 discusses how the impact of this phenomenon can be alleviated in practice.

5 Conclusions and future work

Although performed testing was limited in scope, one can draw some conclusions about the efficiency of the proposed approach. Computing shortest violation traces can facilitate the debugging process and save a lot of designer's time, since in many cases the first computed violation trace is much longer than a shortest one. According to the experimental results, for large problem instances it can reduce the number of new variables in the formula by two–three orders of magnitude, and achieve almost optimal reduction in the number of clauses and literals, i.e., the length of the CNF formula corresponding to the threshold constraint was surprisingly close to that for a single multiple-input $\vee$-gate!

The possible directions for future research include using a Boolean minimiser to derive short formulae not only for half-adder and full-adder cells but also for adders with a small number of inputs, and exploiting the structure of the prefix to reduce the size of other pseudo-Boolean constraints encountered when dealing with various model checking problems.

Acknowledgements The author would like to thank Keijo Heljanko for fruitful discussions. This research was supported by an EC IST grant 511599 (Rodin).

References

1. J. C. Corbett: Evaluating Deadlock Detection Methods for Concurrent Software. *IEEE Transactions on Software Engineering* 22(3) (1996) 161–180.
2. J. Esparza: Decidability and Complexity of Petri Net Problems — an Introduction. *Lectures on Petri Nets I: Basic Models.* LNCS 1491 (1998) 374–428.
3. M. Garey and D. Johnson: *Computers and Intractability — A Guide to the Theory of NP-completeness.* Freeman (1979).
4. J. E. Hopcroft and R. M. Karp: An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs. *SIAM Journal on Computing* 2(4) (1973) 225–231.
5. V. Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings.* School of Comp. Sci., Univ. of Newcastle (2003).
6. V. Khomenko: Computing Shortest Violation Traces in Model Checking Based on Petri Net Unfoldings and SAT. TRep. CS-TR-841, School of Comp. Sci., Univ. of Newcastle (2004). URL: `http://homepages.cs.ncl.ac.uk/victor.khomenko/home.formal/papers/papers.html`
7. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of *CAV'1992*, LNCS 663 (1992) 164–174.
8. S. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik: Chaff: Engineering an Efficient SAT Solver. Proc. of *DAC'2001*, ASME Techn. Publ. (2001) 530–535.
9. I. Wegener: *The Complexity of Boolean Functions.* Wiley-Teubner Series in Computer Science (1987).
10. L. Zhang and S. Malik: The Quest for Efficient Boolean Satisfiability Solvers. Proc. of *CAV'2002*, E. Brinksma and K. G. Larsen (Eds.). LNCS 2404 (2002) 582–595.