Project DEPLOY
Grant Agreement 214158

"Industrial deployment of advanced system engineering methods for high productivity and dependability"

DEPLOY Deliverable D54

D13.6 Final Report "Enlarged EU"

Public Document

April 2012

http://www.deploy-project.eu

Contributors:

Luigia Petre	Aabo Akademi University (Aabo)
Kaisa Sere	Aabo Akademi University (Aabo)
Gheorghe Stefanescu	University of Bucharest (UniBuc)
Denisa Diaconescu	University of Bucharest (UniBuc)
Ioana Leustean	University of Bucharest (UniBuc)
Radu Gramatovici	University of Bucharest (UniBuc)
Michael Leuschel	University Duesseldorf (UDUS)
Daniel Plagge	University Duesseldorf (UDUS)
Florentin Ipate	University of Pitesti (UniPit)
Alin Stefanescu	University of Pitesti (UniPit)
Ionut Dinca	University of Pitesti (UniPit)
Raluca Lefticaru	University of Pitesti (UniPit)
Cristina Tudose	University of Pitesti (UniPit)
Laurentiu Mierla	University of Pitesti (UniPit)
Sebastian Wieczorek	SAP

Editors:

Alin Stefanescu	University of Pitesti (UniPit)
Sebastian Wieczorek	SAP

Reviewers:

Luigia Petre	Aabo Akademi University
Michael Poppleton	Southampton University

Abstract

This deliverable summarizes the work conducted by the partners in the DEPLOY extension called "DEPLOY Enlarged EU". As a result of this extension, two new partners from Romania, University of Bucharest and University of Pitesti, joined the DEPLOY consortium. The work started in June 2010 and continued until the end of the DEPLOY project in April 2012. The objective of this deliverable is to report on the research performed in this project extension and on the integration of the new partners in the project.

Contents

Chapter 1

Executive summary

On June 1, 2010 two new partners, University of Bucharest (UniBuc) and University of Pitesti (UniPit) from the enlarged EU joined the DEPLOY IP to contribute to the second half of the project on industrial deployment of system engineering methods providing high dependability and productivity.

In this short introduction, we summarize the main achievements and performed activities. The technical details are provided in the following chapters.

Technical contributions. As originally planned, the main new technical contributions of DEPLOY-Enlarged EU followed two strands of work:

- Based on the flexible structuring mechanisms of the register-voice interactive systems formalism, UniBuc and Aabo studied their applicability to Event-B in two directions. First, we have investigated novel composition mechanisms in Event-B as well as techniques to alleviate the discharging of proof obligations in Event-B and second, we have proposed finer-grained synchronization between parallel processing with a significant impact on the Event-B modeling of time. The details are described in Chapter 2.
- UniPit studied together with partners from SAP and UDUS, model-based testing techniques (MBT) for Event-B, using automata learning, constraint solving, and genetic algorithms. Details can be found in Chapter 3.

In order to better understand the work done to satisfy the requirements and goals in the extended DEPLOY's Description of Work (DoW), we provide in Table 1.1 side-by-side the set of extended tasks (Tasks 7.1, 8.5, 9.10) and a new one (Task 12.4) together with the actions performed during the project.

Research papers. To ensure proper dissemination, all results of the research were submitted as papers to different venues (journals, conferences, and work-

Extended/new tasks in DoW	**Its realization in DEPLOY "Enlarged EU"**
Extension of **Task 7.1 Composition** from WP7: *[...] We will exploit the interaction systems (IS) techniques, including their stream-based composition operators, to further increase the power of the formalism. Particular emphasis will be on using structured interaction composition operators. To this end we will express stream-based abstract temporal data on interaction interfaces to support decomposition of systems into arbitrary small components, which can be further aggregated, applying the interactive system composition operators, to reconstruct the initial systems.*	In order to exploit the IS techniques in Event-B, we worked towards a tight integration level of the two formal frameworks, up to a level where key features of each formalism can be easily translated into the other. Since a notion of refinement is a key feature in Event-B but not in IS and structured interaction composition is a key operation in IS but not in Event-B, we studied them in conjunction with refinement-preserving mappings from one formalism to the other and vice versa. The technical details are provided in Section 2.1.
Extension of **Task 8.5 Modelling and Analysis of Real-TIme Systems** from WP8: *[...] We view the patterns of temporal data as dual to the patterns of memory states. Temporal pointers (specifying the starting time of data on streams) will be used to represent temporal data and the IS interaction composition operators will be used to deal with real-time constraints.*	The above integration between IS and Event-B is also used to address Task 8.5, by mapping the temporal notions to Event-B. More precisely, the temporal features of IS are modeled by 'temporal pointers', which appear in the interfaces of the IS interaction composition. This is possible even in the presence of refinement. The technical details are given in Section 2.2.
Extension of **Task 9.10 Model-based Testing** from WP9: *[...] Techniques to optimize and minimize test cases will also be developed. Model-based testing (MBT) is of particular interest to the deployment in WP4 and feedback from WP4 will be important in order to fine-tune the techniques. The used MBT approach will be enhanced with search-based approaches improving test data generation which is particularly important for business information systems. We will use evolutionary approaches to bring a significant contribution to test data generation, by applying different genetically-inspired algorithms for generating various data mutants that have as their evolution goal the satisfaction of the data constraints. This work complements the directed model checking techniques, with the potential of increasing event coverage required by MBT based Event-B specification, as well as decreasing the effort of exploration of the large state spaces induced by the Event-B specification.*	In DEPLOY "Enlarged EU" we succeeded to achieve all the proposed goals and even more. First, we indeed devised and implemented methods of test suite optimization based on genetically-inspired multi-objective optimizations (Section 3.4). Then, we proposed an MBT procedure, i.e., test case generation, by a new approach using automata learning on bounded feasible sequences. As a by-product, we also obtain finite state approximations of the state space of an Event-B model. The method is well integrated with the notion of Event-B refinement (Section 3.1). Without being originally planned, we extended the MBT method to work also with decomposed Event-B models (Section 3.2). Complementing a constraint-based approach, genetically-inspired algorithm were used to generate test data for a given test case (Section 3.3). The implementation of the above is done in a Rodin plug-in and was tested on many Event-B models (Section 3.5).
New **Task 12.4 Introduction of New Partners** from WP12: *Enlarging of the DEPLOY project by the two new partners will require some managerial synchronisation and integration on technical, administrative and financial activities.*	Both universities UniBuc and UniPit were successfully integrated into the project in a timely manner. The collaboration between the team members was also smooth. More details are provided at the end of this executive summary.

Table 1.1: Contributions of DEPLOY Enlarged EU according to the DoW

shops). The result are the following eleven papers given in the bibliography at the end of this section:

[1], **[2]**, **[3]**, **[4]**, **[5]**, **[6]**, **[7]**, **[8]**, **[9]**, **[10]**, **[11]**.

Moreover, the Romanian partners contributed to the following DEPLOY deliverables and also the "D-Book" planned for the end of the project:

D32, **D42**, **D43**, **D44**, **D45**, **D53**, **D54** and a "**D-book**" chapter.

A Rodin plug-in implementing MBT techniques for Event-B was released:

`http://wiki.event-b.org/index.php/MBT_plugin`

Dissemination activities. The results were regularly presented by the members of DEPLOY Enlarged EU at several venues such as seminars, conferences, workshops, tutorials. A detailed list is provided below:

- October 2010 (Dagstuhl, Germany): presentation of MBT in DEPLOY at the Dagstuhl Seminar.
  Speakers: Sebastian Wieczorek (SAP) and Alin Stefanescu (UniPit)

- December 2010 (Timisoara, Romania): tutorial on DEPLOY, Rodin and MBT at a FP7 Training on Software Services.
  Speaker: Alin Stefanescu (UniPit)

- March 2011 (Sheffield, UK): invited talk on MBT in the Department of Computer Science of University of Sheffield.
  Speaker: Florentin Ipate (UniPit)

- March 2011 (Berlin, Germany): presentation at the 4th International Workshop on Search-Based Software Testing (SBST'11).
  Speaker: Alin Stefanescu (UniPit)

- April 2011 (Turku, Finland): lecture on the UniBuc-Aabo research cooperation at Aabo University.
  Speakers: Luigia Petre (Aabo) and Gheorghe Stefanescu (UniBuc)

- May 2011 (London, UK): invited talk on MBT in DEPLOY at 13th CREST[1] Open Workshop on Future Internet Testing (FITTEST) and Search Based Software Engineering (SBSE).
  Speaker: Alin Stefanescu (UniPit)

[1] CREST: Centre for Research on Evolution, Search and Testing at University College London

- June 2011 (Kuantan, Malaysia): presentation at the 2nd International Conference on Software Engineering and Computer Systems (ICSECS'11).
  Speaker: Alin Stefanescu (UniPit)
- June 2011 (Bucharest, Romania): presentation of MBT in DEPLOY at a Romanian software company called SOFTWIN.
  Speaker: Alin Stefanescu (UniPit)
- June 2011 (Thessaloniki, Greece): invited talk on MBT in DEPLOY at the South-East European Research Centre (SEERC) in Thessaloniki.
  Speaker: Florentin Ipate (UniPit)
- November 2011 (Helsinki, Finland): presentation of MBT in DEPLOY at the lab seminar at the Computer Science Department of Aalto University.
  Speaker: Alin Stefanescu (UniPit)
- February 2012 (Fontainebleau, France): MBT demo at the DEPLOY Federated Event.
  Speaker: Laurentiu Mierla (UniPit)
- May 2012 (Edinburgh, UK): presentation of MBT in DEPLOY at the weekly Computer Science Department seminar at the University of Edinburgh.
  Speaker: Alin Stefanescu (UniPit)
- June 2012 (Pisa, Italy): presentations at 3rd International Conference on Abstract State Machines, Alloy, B, and Z (ABZ'12) and 9th International Conference on Integrated Formal Methods (iFM'12).
  Speakers: Alin Stefanescu (UniPit) and Gheorghe Stefanescu (UniBuc)

Integration of the new partners within the existing consortium. Integration of the new partners was successful. They were brought up-to-speed using a dedicated training session. Further on, the collaboration with DEPLOY partners was initiated and supported not only by participation of the project-wide meetings (in Nice, Newcastle, Zurich, Fontainbleau) or executive meetings, but also face-to-face meetings. Moreover, UniPit took part in the WP9 bi-weekly teleconferences. Last but not least, young researchers, especially PhD students, were involved in the research and the project meetings.

Below we list the face-to-face short visits (from one to four weeks) and meetings (one-two days) of Romanian partners with DEPLOY partners:

- June 2010: Kick-off meeting for "DEPLOY Enlarged EU" in Darmstadt.
- July 2010: Two-days training session in Bucharest, Romania with Thai Son Hoang (ETH Zurich) and Stefan Hallerstede (UDUS) as trainers.

- September 2010: UniPit visited SAP in Darmstadt.
- November 2010: UniPit visited SAP in Darmstadt.
- March-April 2011: UniBuc visited Aabo in Turku.
- March 2011: UniPit visited SAP in Darmstadt.
- September 2011: UniPit visited UDUS in Dusseldorf.
- September 2011: UniPit visited SAP in Darmstadt.
- November-December 2011: UniBuc visited Aabo in Turku.

Conclusions. At the end of the project, we evaluate the project extension "Enlarged EU" as a successful part of the DEPLOY project. The outcome contributed to the development of Event-B method, both theoretically (several papers validated by the scientific community via peer-reviewing) and practically (a Rodin plug-in thoroughly tested on the Event-B models from the DEPLOY repository). The collaboration between the researchers was also very good and the results were disseminated to a wide audience. The new partners, including the young researchers, highly appreciated the experience to work in such a large European project on state-of-the art research and concrete industrial requirements. As for the future, there are plans to further pursue research in the Event-B area and maintain the existing tooling. The funding for the research on MBT after DEPLOY will be ensured by two recently acquired projects: UDUS is part of the FP7 project *ADVANCE* (2011-2014 – http://www.advance-ict.eu/) and UniPit has a new Romanian grant *MuVet* (2012-2014 – grant no. PN-II-ID-PCE-2011-3-0688). Both projects have MBT for Event-B as an explicit task.

Papers produced in DEPLOY Enlarged EU

[1] Denisa Diaconescu, Ioana Leustean, Luigia Petre, Kaisa Sere, and Gheorghe Stefanescu. Refinement preserving translation from Event-B to register-voice interactive systems. Technical Report no. 1028, TUCS (Turku Center for Computer Science), 51 pp., December 2011. `http://tucs.fi`

[2] Denisa Diaconescu, Ioana Leustean, Luigia Petre, Kaisa Sere, and Gheorghe Stefanescu. Refinement preserving translation from Event-B to register-voice interactive systems. In *Proc. of Int. Conf. on Integrated Formal Methods (iFM'12)*, LNCS, vol. 7321, pp. 221–236. Springer, 2012.

[3] Denisa Diaconescu, Ioana Leustean, Luigia Petre, Kaisa Sere, and Gheorghe Stefanescu. Refinement of Structured Interactive Systems. Submitted to *Formal Methods (FM'12) conference*. March 2012.

[4] Radu Gramatovici, Luigia Petre, Kaisa Sere, Alin Stefanescu, and Gheorghe Stefanescu. Syncronization in Timed-Interactive Systems. Submitted to *19th International Symposium on Temporal Representation and Reasoning (TIME'12)*. April 2012.

[5] Alin Stefanescu, Sebastian Wieczorek, and Matthias Schur. Message choreography modeling – a domain-specific language for consistent enterprise service integration. Conditionally accepted at *Software and Systems Modeling (SoSyM)* journal, 2012.

[6] Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose. Towards search-based testing for Event-B models. In *Proc. of 4th Workshop on Search-Based Software Testing (SBST'11)* from ICSTW'11, pp.194-197. IEEE Computer Society, 2011.

[7] Ionut Dinca, Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose. Test data generation for Event-B models using genetic algorithms. In *Proc. of 2nd International Conference on Software Engineering and Computer Systems (ICSECS'11)*, volume 181 of *CCIS*, pp. 76–90. Springer, 2011.

[8] Ionut Dinca. Multi-objective test suite optimization for Event-B models. In *Proc. of Int. Conf. on Informatics Engineering and Information Science (ICIEIS'11)*, vol. 251 of *CCIS*, pp. 551–565. Springer, 2011.

[9] Ionut Dinca, Florentin Ipate, Laurentiu Mierla, and Alin Stefanescu. Learn and test for Event-B – a Rodin plugin. In *3rd International Conference on Abstract State Machines, Alloy, B, and Z (ABZ'12)*. LNCS, vol. 7316, pp. 361–364. Springer, 2012.

[10] Florentin Ipate, Ionut Dinca, and Alin Stefanescu. Model learning and test generation using cover automata. Submitted to *IEEE Transactions on Software Engineering* journal. January 2012.

[11] Ionut Dinca, Florentin Ipate, and Alin Stefanescu. Model learning and test generation for Event-B decomposition. Submitted to *5th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'12)*. April 2012.

Chapter 2

Event-B and Register-Voice Interactive Systems (rv-IS)

Contributors: Denisa Diaconescu (UniBuc), Radu Gramatovici (UniBuc), Ioana Leustean (UniBuc), Luigia Petre (Aabo), Kaisa Sere (Aabo), Alin Stefanescu (UniPit), and Gheorghe Stefanescu (UniBuc)

The goal of this strand of work was to find ways of integrating the Event-B and Register-Voice Interactive Systems (rv-IS) formalisms. The following plan was pursued:

- Define a notion of refinement in rv-IS models based on a combination of the refinement of state-based systems and Broy-like refinement of dataflow-based interactive systems.

- Define a translation EB2IS from Event-B models to structured rv-IS models.

- Prove the translation EB2IS preserves refinement (i.e., refined Event-B models translate into refined rv-IS models, the latter notion being the proposed rv-IS refinement).

- Use one of the known translations to pass from structured rv-IS models to unstructured rv-IS models.

- Define a refinement preserving translation IS2EB from unstructured rv-IS models to Event-B models.

- Use these translations EB2IS and IS2EB to: (1) enrich Event-B with rv-IS structural operators and the associated decomposition techniques; (2) get tool support to develop and analyze rv-IS models.

The material in this chapter is based on several papers already accepted or under review: [65], [66], [67], [68].

2.1 Work on Event-B (de)composition

From the above working plan, the first investigations developed a refinement notion on rv-IS models and a refinement preserving translation from Event-B to rv-IS models. Event-B is a state-based formalism dedicated to the refinement-based development of correct parallel and distributed systems. The register-voice interactive systems (rv-IS) formalism is a recent structural approach for developing software systems using both structural state-based as well as interaction-based composition operators. One of the most interesting feature of the rv-IS formalism is the structuring of the components interactions. Our aim is to study whether a more (rv-IS inspired) structured approach of an interactive, modular system has any effect on the correct development as designed in Event-B. More precisely, we are interested in uncovering whether the proof obligations are significantly eased when a certain structure is assumed in the model. Towards this end, we need to develop an integration of Event-B and rv-IS, that would ultimately also imply tool support to develop and analyse rv-IS models. In this section we introduce the notion of refinement in the rv-IS formalism and based on it we propose a refinement-based translation from Event-B to rv-IS, also exemplified with a file transfer protocol modelled in both formalisms.

The register-voice interactive systems formalism (rv-IS) is a structural approach for developing software systems using both state- and interaction-based composition operators. The aim of our study is to integrate the Event-B and rv-IS formalisms up to a level where the key features of each formalism can be easily translated into the other. The contribution of the section is threefold. First, we introduce a scenario-based notion of refinement in rv-IS, extending the trace-based definition of refinement of classical sequential systems. Next, we present a refinement preserving translation EB2IS from Event-B models to structured rv-IS models. Finally, we argue our translation by analyzing an example: we present a refining process for modeling a simple file transfer protocol in Event-B and we show the associated set of refined structured rv-IS models.

One possibility to define the correctness of the EB2IS translation is via the trace semantics. For an Event-B model M, one can define the traces associated to all possible runnings of the model. On the other hand, one can consider the translated rv-IS model M' and the associated running scenarios; starting with these scenarios and using the flattening operator, a set of traces may be associated to the translated model M'. The correctness problem for the EB2IS translation must show that the translation preserves the associated traces, up to state stuttering and a hiding of the auxiliary variables and modules introduced by the translation. Without giving a formal proof that the translation is correct according to this definition, the examples included in Subsection 2.1.6 provide evidence in this direction

2.1.1 Introduction

This section presents a refinement preserving translation from Event-B [3, 2, 29, 37, 38, 39, 16, 18] to register-voice interactive systems (rv-IS) [55, 56, 57, 45, 33, 34, 51]. The register-voice interactive systems formalism (rv-IS) is a recent structural approach for developing software systems using both structural state- and interaction-based composition operators.

Interactive computation is an important computer science topics. Often, the term is related to HCI (Human-Computer Interaction), the particular case when one of the interacting entities is human. While able to deal with such cases as well, our approach is more process-process interaction oriented. In this latter case, there are already many successful formalisms, including models as Petri nets [58], process algebra [24], π-calculus [41], dataflow networks [26, 28], event structures [62], temporal logics formalisms [44, 40], etc. The approach used in the present section integrates a dataflow like interaction model with a classical state-based computation model.

Specifically, we use the register-voice interactive systems formalism with the following characteristic features.

1. The model is based on *register-voice interactive systems (rv-IS)* [55, 56]. This class of models includes register machines and dataflow networks, is space-time invariant, is compositional, and may describe computations extending both in time and in space. The specific application area for rv-IS models is the class of open, interactive systems.

2. The formalism includes a *programming language* style which uses novel techniques for the syntax and the semantics to support the computation in the space paradigm. To this end, it uses *rv-programs* [55, 56] whose syntax and operational semantics are based on *finite interactive systems (FISs)* and running *scenarios* [54, 55, 56].

3. For *specification* of the rv-systems the formalism uses relations between input and output interfaces [55, 56]. These interfaces are complex spatial and temporal structures built up from registers and voices, see, for instance the interfaces used in Agapia v0.1 programming langage [33].

The aim of our study is to integrate the Event-B and rv-IS formalisms up to a level where the key features of each formalism can be easily translated into the other.

The contribution of this section is threefold. First, we introduce a scenario-based notion of refinement in rv-IS models, extending the trace-based definition of the refinement of classical sequential systems. Next, we present a refinement preserving translation EB2IS form Event-B models to structured rv-IS models. Finally, we argue our translation by analyzing an example: we present a refining process for modeling a simple file transfer protocol in Event B and we show the associated set of refined structured rv-IS models.

To summarize, roughly half of the above roadmap is, at least partially, dealt with in this section.

The chapter is organized as follows. The first two sections contain brief introductions to Event-B and register-voice interactive systems. The next section tackle the scenario equivalence problem, in particular defining a sub-scenario stuttering equivalence. The core section is Section 2.1.5 where a scenario based notion of refinement for register-voice interactive systems is introduced. The next section describes a general translation from Event-B models to rv-IS models. Section 2.1.7 presents a detailed example dealing with a "file transfer protocol": the initial model and two refined versions are first developed in Event-B, then translated into the rv-IS framework; then, it is shown that the translation preserves refinement. A brief section with final comments and the references conclude the section.

2.1.2 Event-B

Introduction to Event-B

Event-B Language. In Event-B, a system specification (model) is defined using the notion of a *machine* [48] operating on an *abstract state*. Such a machine encapsulates the model state, represented as a collection of model variables, and defines operations on this state. Thus, it describes the behavior of the modeled system, also referred to as the dynamic part. A machine may also have an accompanying component, called *context*, which contains the static part of the system. A context can include user-defined carrier sets, constants and their properties, which are given as a list of model axioms. The general form of an Event-B model is illustrated in Fig. 2.1. The relationship between a machine and its accompanying context is expressed by the keyword *Sees*, denoting a structuring technique that allows the machine access to the contents of the context.

A machine is uniquely identified by its name M. The state variables, v, are declared in the **Variables** clause and initialized by the $Init$ event. The variables are strongly typed by the constraining predicates I given in the **Invariants** clause. The invariant clause may also contain other predicates defining properties that should be preserved over the state of the model.

The dynamic behavior of the system is defined by a set of atomic events specified in the **Events** clause. Generally, an event can be defined as follows:

$$evt \;\widehat{=}\; \textbf{any } vl \textbf{ where } g \textbf{ then } S \textbf{ end},$$

where the variable list vl contains new local variables (parameters) of the event, the guard g is a conjunction of predicates over the state variables v and vl, and the action S is an assignment to the state variables.

The occurrence of events represents the observable behavior of the system. The event guard defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a parallel composition of assignments. The assignments can be either deterministic or non-deterministic. A deterministic assignment, $x := E(x, y)$, has the standard syntax and meaning. A non-deterministic assignment is denoted either as $x :\in Set$, where Set is a set of values, or $x :| \; P(x, y, x')$, where P is a predicate relating initial values of x, y to some final value of x'. As a result of such a non-deterministic assignment, x can get any value belonging to Set or according to P.

Event-B Semantics. The semantics of Event-B actions is defined using so-called before-after (BA) predicates [3, 48]. A before-after predicate describes a relationship between the system states before and after execution of an event, as shown in Fig. 2.2. Here x and y are disjoint lists (partitions) of state variables, and x', y' represent their values in the after-state. A before-after predicate for events is constructed as follows:

$$BA(evt) \;=\; \exists vl.\, g \,\wedge\, BA(S).$$

Here vl stand for some local variables of the event evt, the guard g is a conjunction of predicates over the state variables v and vl, and the action S is an assignment to the state variables.

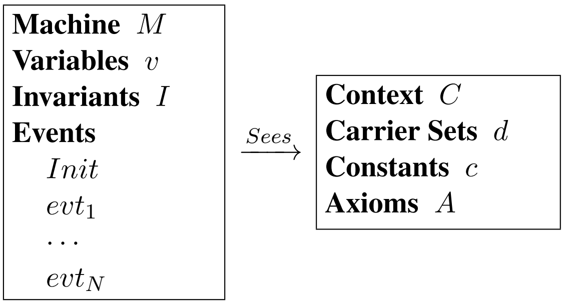


Figure 2.1: A machine M and a context C in Event-B

Action (S)	$BA(S)$
$x := E(x,y)$	$x' = E(x,y) \ \wedge \ y' = y$
$x :\in Set$	$x' \in Set \ \wedge \ y' = y$
$x :\mid P(x,y,x')$	$x' \in \{ \, v \mid P(x,y,v) \, \} \ \wedge \ y' = y$

Figure 2.2: Before-after predicates

The semantics of a whole Event-B model is formulated as a number of *proof obligations*, expressed in the form of logical sequents and detailed in [3].

System Development. Event-B employs a top-down refinement-based approach to formal system development. Development starts from an abstract system specification that models some of essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into an abstract specification. These new events correspond to stuttering steps that are not visible in the abstract specification. We call such model refinement as *superposition refinement*. Moreover, Event-B formal development supports data refinement, allowing us to replace some abstract variables with their concrete counterparts. In that case, the invariant of a refined model formally defines the relationship between the abstract and concrete variables; this type of invariants are called *gluing invariants*.

To verify the correctness of a refinement step, we need to prove a number of proof obligations for a refined model, also detailed in [3].

The Event-B refinement process allows us to gradually introduce implementation details, while preserving functional correctness during stepwise model transformation. The model verification effort and, in particular, automatic generation and proving of the required proof obligations, are significantly facilitated by the provided tool support – the RODIN platform [2, 59, 48, 4].

Let us note here the quintessential feature of Event-B and its associated RODIN platform. Modeling in Event-B is semantically justified by proof obligations. Every update of a model generates a new set of proof obligations in the background. It is this interplay between modeling and proving that sets Event-B apart from other formalisms. Without proving the required obligations, we cannot be sure of correctness of a model. The proving effort thus encourages the developer to structure formal model development in such a way that manageable proof obligations are generated at each step. This leads to very abstract initial models so that we can gradually introduce into a system model various facets of the system. Such a development method fits well when we have to describe complex algorithms.

2.1.3 Register-voice interactive systems

Interactive computation is an important computer science topics. Often, the term is related to HCI (Human-Computer Interaction), the particular case when one of the interacting entities is human. While able to deal with such cases as well, our approach is more process-process interaction oriented. In this latter case, there are many successful formalisms, including: (1) true concurrency models like Petri nets, dataflow networks, event structures, etc.; or (2) interleaving models as process algebra, π-calculus, temporal logics formalisms, communicating automata, etc. The approach used in the present section integrates this type of interaction models with classical state-based computation models.

Specifically, we use the model, the core programming language, the specification formalism and the analysis techniques used for modeling, programming and reasoning about interactive computing systems that have been developed by the last author and coworkers in the last years, see [56, 57, 45, 33, 34, 51]. The formalism is built on top of register machines, closing them with respect to space-time duality operator. It has the following characteristic features:

1. The model consists of *register-voice interactive systems (rv-IS)*. The class includes register machines, is space-time invariant, is compositional and may describe computations extending both in time and space. Its specific application area is the class of open, interactive systems.

2. The programming language uses novel techniques for its syntax and semantics to support computation in the space paradigm. To this end, it uses *rv-programs* whose syntax and operational semantics are based on *finite interactive systems (FISs)* and running *scenarios*.

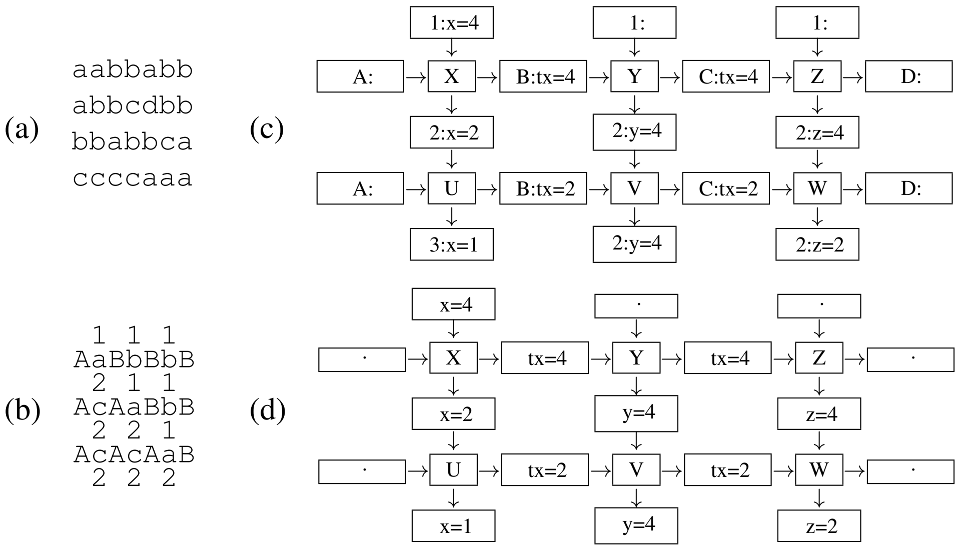
3. The *specification* of the rv-systems uses relations between their input and output interfaces. These interfaces are complex spatial and temporal structures built up from registers and voices.

The rest of this subsection includes a glimpse of the approach.

Grids and scenarios.

A *grid* is a two-dimensional rectangular area filled in with letters of a given alphabet. In our interpretation the columns correspond to processes, the top-to-bottom order describing their progress in time. The left-to-right order corresponds to process interaction in a nonblocking message passing discipline. This means, a process sends a message to the right, then it resumes its execution. An example is presented in Fig.2.3(a).

A *scenario* is a grid enriched with data around each letter. The data may be given in an abstract form as in Fig. 2.3(b), or in a more detailed form as in Fig. 2.3(c)&(d). In (a) there are grid letters, only. In (b), additional information is inserted: (1) abstract notations for *(memory) states*, denoted here by numbers and placed at the north and the south borders of the grid letters; and (2) abstract notations for *(interaction) classes*, denoted here by capital letters and placed at the west and the est borders of the letters. Scenarios as in (b) are abstract and are used to check if a grid is recognized by a FIS. In (c) a more concrete scenario is presented where also data are associated to states and classes. These latter type of scenarios are used for representing rv-programs runs. Finally, in (d) a run of a *structured rv-program* is shown: for structured rv-programs, no state and class names are needed, so there are no numbers and capital letters around the grid letters as in (c).



(a)

```
aabbabb
abbcdbb
bbabbca
ccccaaa
```

(b)

```
 1 1 1
AaBbBbB
 2 1 1
AcAaBbB
 2 2 1
AcAcAaB
 2 2 2
```

Figure 2.3: A grid (a), an abstract scenario (b), and concrete scenarios (c) and (d).

Spatio-temporal specifications.

A spatio-temporal specification combines constraints on both spatial and temporal data.

For the spatial data, we use the common data structures and their natural representations in memory.

For representing temporal data we use streams: a *stream* is a sequence of data ordered in time and is denoted as $a_0{}^\frown a_1{}^\frown \ldots$, where $a_0, a_1, \ldots$ are the data laying on the stream at time $0, 1, \ldots$, respectively.

A *voice* is defined as the time-dual of a register. Voices are simple temporal structures, represented on streams, that holds natural numbers. The value of a voice can be modified in a location and then propagated within the system. A voice can be "listened" at various locations, at each location the piece of stream representing the voice displaying a particular value. Voices may be implemented on top of a stream in a similar way registers are implemented on top of a Turing tape, for instance specifying their starting address and their length. Most of usual data structures have natural temporal representations. Examples includes timed booleans, timed integers, timed arrays, timed lists, etc.

The notation $\otimes$ is used for the product of memory states, while $\frown$ for the product of interaction classes; $\mathbb{N}^{\otimes k}$ denotes $\mathbb{N}\otimes \ldots \otimes\mathbb{N}$ (k terms) and $\mathbb{N}^{\frown k}$ denotes $\mathbb{N}^{\frown}\ldots{}^{\frown}\mathbb{N}$ (k terms).

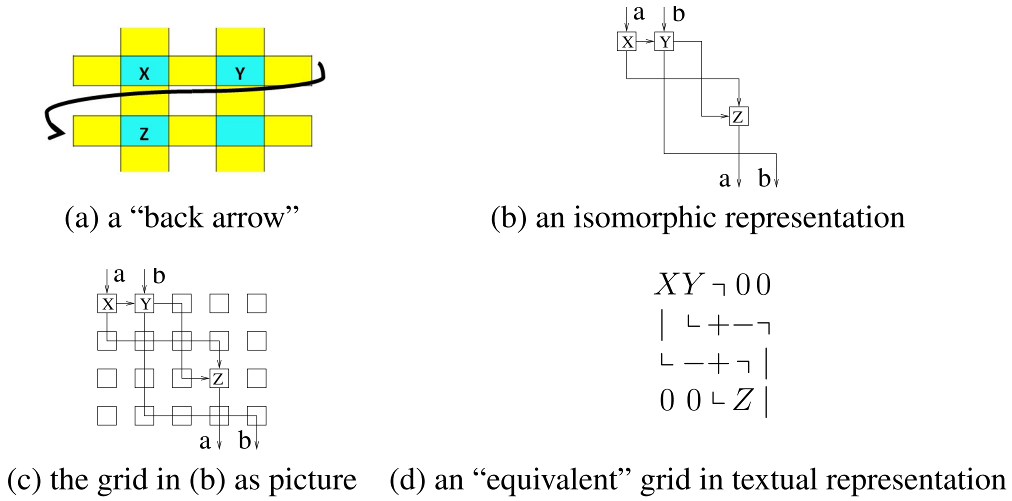
A simple *spatio-temporal specification*

$$S : (m,p) \rightarrow (n,q)$$

is a relation $S \subseteq (\mathbb{N}^{\frown m} \times \mathbb{N}^{\otimes p}) \times (\mathbb{N}^{\frown n} \times \mathbb{N}^{\otimes q})$, where m (resp. p) is the number of input voices (resp. registers) and n (resp. q) is the number of output voices (resp. registers). It can be described as a relation between tuples, the notation being

$$< v \mid r > \rightarrow < v' \mid r' >,$$

where v, v' (resp. r, r') are tuples of voices (resp. registers). More general spatio-temporal specifications may be introduced using complex interface types, not only registers and voices.



(a) a "back arrow"

(b) an isomorphic representation

(c) the grid in (b) as picture

```
XY ┐ 0 0
|  └ + − ┐
└ − + ┐ |
0 0 └ Z |
```

(d) an "equivalent" grid in textual representation

Figure 2.4: Modeling "back arrows"

Examples. We describe below a few very simple specifications. Actually, they give the semantics for the following wiring constants used in Fig. 2.4(c):

$c0 =$, $c1 =$, $c2 =$, $c3 =$, $c4 =$, $c5 =$

The natural relational interpretation of these constants is:

$c0 = \emptyset$ (*empty cell*);
$c1 = \{\langle \mid x\rangle \mapsto \langle \mid x\rangle : \ x \in \mathbb{N}\}$ (*vertical identity*);
$c2 = \{\langle x \mid \rangle \mapsto \langle x \mid \rangle : \ x \in \mathbb{N}\}$ (*horizontal identity*);
$c3 = \{\langle \mid x\rangle \mapsto \langle x \mid \rangle : \ x \in \mathbb{N}\}$ (*speaker*, or *space-to-time converter*);
$c4 = \{\langle x \mid \rangle \mapsto \langle \mid x\rangle : \ x \in \mathbb{N}\}$ (*recorder*, or *time-to-space converter*);
$c5 = \{\langle x \mid y\rangle \mapsto \langle x \mid y\rangle : \ x, y \in \mathbb{N}\}$ (*double identity*, or *cross*).

These basic specifications are extended in a straightforward way to the case of complex data associated to a wire, so x above may have any data type not only be a natural number.

Specifications may be composed horizontally and vertically, as long as their types agree. For instance, given two specifications $S_1 : (m_1, p_1) \to (n_1, q_1)$ and $S_2 : (m_2, p_2) \to (n_2, q_2)$, their *horizontal composition* $S_1 \# S_2$ is defined only if $n_1 = m_2$; the type of $S_1 \# S_2$ is $(m_1, p_1 + p_2) \to (n_2, q_1 + q_2)$ and the result is the expected relational composition via connecting voices.

The comments above were restricted to register and voice data types. However, the approach can be easily extended to deal with complex spatial and temporal data types, for instance the interface types used in Agapia v0.1 language [33].

Syntax of structured rv-programs.

The *type* of a *structured rv-program* P, denoted by

$$P : (w(P), n(P)) \to (e(P), s(P)),$$

collects the types at the west, north, east and south borders of its scenarios. In general, these are relatively complex types built up from atomic boolean and integer types - see the concrete types used in Agapia v0.1 programming language [33].

Syntax of structured rv-programs is defined as follows:

```
P ::= X | P % P | P # P | P $ P | if(C) then {P} else {P}
        | while_t(C) {P} | while_s(C) {P} | while_st(C) {P}
```

The starting blocks for the construction of structured rv-programs are called *modules*. The syntax of a module is given as follows:

```
module module_name
{listen temporal_variables}{read
spatial_variables}{
  code
}{speak temporal_variables}{write
spatial_variables}
```

where the `read` (resp. `listen`) instruction collects the spatial (resp. temporal) input and the `write` (resp. `speak`) instruction returns the spatial (resp. temporal) output. The `code` consists in instructions similar to the C code.

The instruction that reads the input or returns the output are sometimes omitted by taking into consideration the following conventions: any variable not declared in the code will be considered an input variable and any variable that appears in the code will be considered an output variable.

The operations on structured rv-programs are briefly described below. More details and examples may be found in [55, 56, 33, 34].

1. **Composition:** Due to their two dimensional structure, programs may be composed horizontally and vertically, as long as their types agree. They can also be composed diagonally by mixing the horizontal and vertical composition.

   (a) For two programs $\mathtt{P_i} : (w_i, n_i) \to (e_i, s_i)$, $i = 1, 2$, the *horizontal composition* $\mathtt{P}_1\#\mathtt{P}_2$ is well defined only if $e_1 = w_2$; the type of the composite is $(w_1, n_1 \otimes n_2) \to (e_2, s_1 \otimes s_2)$.

   (b) Similarly, the *vertical composition* $\mathtt{P}_1\%\mathtt{P}_2$ is defined only if $s_1 = n_2$; the type of the composite is $(w_1{}^\frown w_2, n_1) \to (e_1{}^\frown e_2, s_2)$.

   (c) The *diagonal composition* $\mathtt{P}_1\$\mathtt{P}_2$ is a derived operation - it connects the east border of $\mathtt{P}_1$ to the west border of $\mathtt{P}_2$ and the south border of $\mathtt{P}_1$ to the north border of $\mathtt{P}_2$; it is defined only if $e_1 = w_2$ and $s_1 = n_2$; the type of the composite is $(w_1, n_1) \to (e_2, s_2)$.

2. **If:** For the "if" operation, given two programs with the same type $\mathtt{P}, \mathtt{Q} : (w, n) \to (e, s)$, a new program $\mathtt{if(C)\ then}\ \{\mathtt{P}\}\ \mathtt{else}\ \{\mathtt{Q}\} : (w, n) \to (e, s)$ is constructed, for a condition `C` involving both, the temporal variables in w and the spatial variables in n.

3. **While:** There are various possible extensions of the while statement.

   (a) For a program with dummy interaction interface $\mathtt{P} : (0, n) \to (0, n)$, a *temporal while* is defined as a natural extension of the usual "while" statement, namely $\mathtt{while\_t(C)}\{\mathtt{P}\}$, where `C` is a condition on the spatial variables in n; the type of the result is $(0, n) \to (0, n)$.

(b) Similarly, for a program with a dummy state interface $\mathtt{P} : (w, 0) \to (w, 0)$, a *spatial while* is defined as while_s(C){P}, where C is a condition on the temporal variables in w; the type of the result is $(w, 0) \to (w, 0)$.

(c) A *spatio-temporal while* may be defined for a program $\mathtt{P} : (w, n) \to (w, n)$, namely while_st(C){P}, where C is a condition on the temporal variables in w and the spatial variables in n; the type of the result is $(w, n) \to (w, n)$.

The temporal while statement may be extended to the case of programs with nonempty interaction interfaces. In such a case, as the while loop may be executed an indefinite number of times depending on its current data, the west interface has to provide as many interaction data as needed. One may introduce an appropriate construct forall_t{P} or repeat_t{P}, for a $\mathtt{P} : (w, n) \to (e, n)$, which repeats the application of P if its temporal input/output interfaces fits. A similar extension may be applied to the spatial while. Finally, let us notice that no such problems appear in the case of the spatio-temporal while.

Operational semantics of structured rv-programs.

The operational semantics is given in terms of scenarios. Scenarios are built up with the following procedure:

1. Each cell of the associated grid has as label a module name.

2. An area around a cell may have additional information. For example, if a cell has the information $\mathtt{x} = 2$, that means that in that area x is updated to be 2.

3. The scenario is built from the current rv-program by reducing it to simple compositions of spatio-temporal specifications w.r.t. the syntax of the program, until we reach basic blocks, e.g. modules.

We will explain better the operational semantics by considering an example. The following is a structured rv-program **Perfect** which verifies if a number n is perfect:

```
(I1 # I2 # I3) % while_t(x > 0){P # D # M}
```

The modules are listed in Table 2.1.

In our rv-IS program we can imagine that we have three processes: one generates all the number in the set $\{n/2, \ldots, 1\}$ (module P), one checks if a number is a divisor of n (module D) and the last one updates a variable z (module M).

```
module I1
{listen nil}{read n}{
  tn:tInt; x:Int;
  tn = n; x = n/2;
}{speak tn}{write x}
```

```
module I2
{listen tn}{read
nil}{
  y:Int;
  y = tn;
}{speak tn}{write y}
```

```
module I3
{listen tn}{read
nil}{
  z:Int;
  z = tn;
}{speak nil}{write z}
```

```
module P
{listen nil}{read x}{
  td:tInt;
  tx = x; x = x-1;
}{speak tx}{write x}
```

```
module D
{listen tx}{read y}{
  if(y % tx !=0){
    tx = 0;};
}{speak tx}{write y}
```

```
module M
{listen tx}{read z}{
  z = z - tx;
}{speak nil}{write z}
```

Table 2.1: The modules of the **Perfect** rv-IS program

Modules `I1`, `I2` and `I3` are used for initializations. At the end of the program, if the variable z is 0, then the number n is perfect.

In order to show how we can construct a scenario for the rv-IS program above let us consider a concrete example for $n = 6$. The scenario for `n=6` is presented in Figure 2.5.

In the first line of the scenario we initialize the processes with the needed informations: module `I1` is reading the value $n = 6$ and provides the first process with $x = 3$ and declare a temporal variant of n, namely $tn = 6$, that will be used by modules `I2` and `I3` for the other initializations; modules `I2` and `I3` use the temporal variable tn for initializating the other two processes with the initial value of n, namely $y = 6$, $z = 6$, respectively.

In the next step, module `P` produces a temporal data $tx = 3$ (tx is equal with the data x of the first process) and decrease x. Module `D` verifies if tx is a divisor of y and, if no, it resets the value of tx to 0. Finally, module `M` decreases the value of z by tx. Notice that module `M` decreases the value of z only with the divisors of the initial variable n. We continue this steps until the variable x becomes 0.

The same computing scenarios may be generating with many other rv-programs. The above program (and explanation) corresponds to the construction of scenarios by rows and it exhibits a kind of parallel imperative programming style. Below we describe another rv-program which constructs the same scenarios by columns, reflecting a dataflow computing paradigm. To this end, a few more terminating

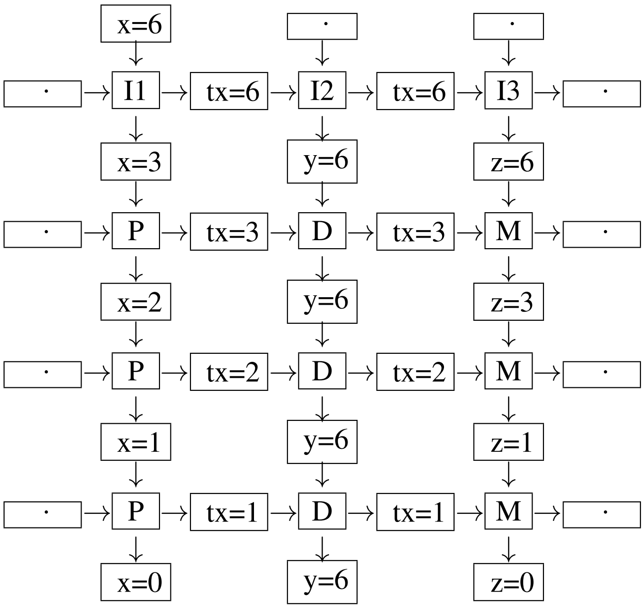


Figure 2.5: A scenario for the **Perfect** rv-IS program

modules have to be used - they are presented in Table 2.2.

The program **Perfect-2** is

```
  (I1 % while_t(x > 0){P} % E1)
# (I2 % while_t(tx != -1){D} % E2)
# (I3 % while_t(tx != -1){M} % E3)
```

The scenarios of this program are the scenarios of the **Perfect** program completed with a bottom row which includes the terminating modules.

```
module E1
{listen nil}{read x}{
  tx = -1;
}{speak tx}{write x}
```

```
module E2
{listen tx}{read y}{

}{speak tx}{write y}
```

```
module E3
{listen tx}{read z}{

}{speak nil}{write z}
```

Table 2.2: The terminating modules of the **Perfect-2** rv-IS program

2.1.4 Scenario equivalence

Before going to introduce a refinement on rv-IS models we have to look carefully to the scenario equality in the presence of wiring constants: identities, space-to-time and time-to-space transformers (e.g., recorders or speakers), etc. These constants have a crucial role in capturing the meaning of state- and class- stuttering in scenarios.

A trace is an abstract representation of a run of a computing system. Operational semantics is often described in terms of "paths", i.e., alternating sequences of states and actions. In some approaches, traces are the projection of paths on states, therefore they are sequences of states. Alternatively, on can project the paths on actions and consider a trace to be a sequence of actions. In this latter case, if the system is deterministic, from the initial state and the trace of actions one can recover the states by applying the actions, hence the action-traces approach is more informal that the state-trace approach. The procedure can be applied also to nondeterministic systems, but in that case it leads to a set of state-traces associated to an action-trace. Generally, it is obvious that from a sequence of states one cannot infer which actions were performed, so the action-traces include more information on the system executions than state-traces.

From this classification point of view, it is worthwhile to mention that the refinement approach is mostly using state-traces.

The notion of running scenarios for rv-programs is a natural extension of the notion of running paths of sequential computing systems. The stuttering relation on traces (i.e., state repetition) is easy to understand. However, defining scenario equality up to a kind of "stuttering equivalence" (i.e., state and class repetition) is more challenging. It is not only the case that processes or communication may stutter, but also more complex phenomena as process or job migration have to be taken into account.

Traces and scenarios

In the following, a dummy interface will be shortly denoted by '.' (in Agapia v0.1 it is denoted by `nil`) and a not-dummy interface by '*'.

Grids vs. scenarios. A scenario is built up from action cells with state and class data around. The basic construction $\mathrm{w}\underset{\mathrm{s}}{\overset{\mathrm{n}}{\boxed{\mathrm{X}}}}\mathrm{e}$ is expanded to larger scenarios applying the gluing rule: the state or class data on the connecting interfaces should be the same.

The additional information around a cell may be included into the cell, for instance as follows $\boxed{\mathrm{w,n,X,e,s}}$. By using such extended letters, scenarios may be represented as simple grids. Actually, they are particular grids where the neighborhood cells satisfy the required gluing conditions. For instance, the following grid with one row and two columns $\boxed{\text{w1,n1,X1,e1,s1}}$ $\boxed{\text{w2,n2,X2,e2,s2}}$ represents a valid scenario only if e1=w2.

To conclude, scenarios may be represented as particular grids over extended alphabets.

Getting traces from scenarios: The flattening operator. The above coding of scenarios by grids may be used to extend the definition of the flattening operator $\flat$ [55] from grids to scenarios. Subsequently, this flattening operator may be used to *associate traces to scenarios* by the following procedure:

1. First, apply the flattening operator $\flat$ to scenarios represented as grids.
2. Than, project the resulting traces on states and classes.

Formally, this defines a function

$$\flat : Sc(S, C, A) \to Tr(S \cup C)$$

where $Sc(S, C, A)$ denotes the scenarios with states in S, classes in C and actions (i.e., cells' labels) in A and $Tr(S \cup C)$ denotes the set of traces over $S \cup C$.

For example, consider the scenario v_1 and the associated grid $g(v_1)$, where $v_1 = \begin{array}{ccccc} & 1 & & 2 & \\ . & X & . & Y & . \\ & 3 & & 4 & \\ a & Z & b & U & c \\ & 5 & & 6 & \\ . & V & . & W & . \\ & 7 & & 8 & \end{array}$ and $g(v_1) = \begin{array}{ccccc} & * & & * & \\ . & (.,1,X,.,3) & . & (.,2,Y,.,4) & . \\ & * & & * & \\ * & (a,3,Z,b,5) & * & (b,4,U,c,6) & * \\ & * & & * & \\ . & (.,5,V,.,7) & . & (.,6,W,.,8) & . \\ & * & & * & \end{array}$. An associated trace in $\flat(g(v_1))$ is $(.,1,X,.,3)(.,2,Y,.,4)(a,3,Z,b,5)(b,4,U,c,6)(.,5,V,.,7)(.,6,W,.,8)$, and by projection on states and classes we get the following trace of state and class changes $(.,1,.,3)(.,2,.,4)(a,3,b,5)(b,4,c,6)(.,5,.,7)(.,6,.,8)$.

Notice that the configuration of the system is not clearly represented in such a trace. Indeed, the trace contains only the changes regarding the local states and classes of the scenario configuration, but not the states and classes positions in the scenario. In order to bypass this shortcoming, one can use explicit localization of the variables. For instance, '6' might be explicitly represented as 'p2.t2.6' meaning that 6 is the state of the 2nd process after the 2nd transaction (or, symmetrically, as 't2.p2.6').

On the other hand, the information is slightly duplicated in these traces. Indeed, at each not-dummy interface which connect two cells, the involved states or classes have the data repeated two times in the trace. One example is class 'b' in the above trace. This should not create problems as traces are normally considered up to a stuttering invariance. However, this stuttering relation is more tricky due to the intrinsic parallelism of the scenario model. Indeed, the stuttering relation may not be not easily detected into a resulting traces as the stuttered states or classes are not necessarily one next to the other in a trace.

The number of traces associated to a scenario is generally huge: for instance a 5×5 scenario has at least 701149020 associated flattened traces (the number may be larger if there are dummy interfaces in the scenario).

Flattening does not commute with refinement. Notice that the flattening operator does not commute with the cells' refinement (i.e., with the substitution of complex grids for cells), this being a main source of difficulties in using trace semantics for concurrent systems.

For example, suppose we start with a high-level communication $v = \begin{array}{ccccc} & * & & * & \\ * & X & * & Y & * \\ & * & & * & \end{array}$, which forces Y to be done after X. Suppose we refine an action Z as "Z_b, followed by Z_c, followed by Z_a", where Z_b, Z_c, Z_a collects the groups of actions before, during, and after communication, respectively. Moreover, we suppose the actions in Z_b (the actions before the communication) and in Z_a (the actions after the communication) are internal actions whose interaction interfaces are nil. Then

v is refined as $v_1 = \begin{array}{ccccc} & * & & * & \\ . & X_b & . & Y_b & . \\ & * & & * & \\ * & X_c & * & Y_c & * \\ & * & & * & \\ . & X_a & . & Y_a & . \\ & * & & * & \end{array}$ and

– $\flat(v) = \{XY\}$, and by refinement we get a unique trace $\{X_bX_cX_aY_bY_cY_a\}$

while

– $\flat(v_1) = \{X_bX_cX_aY_bY_cY_a, X_bX_cY_bX_aY_cY_a, X_bX_cY_bX_aY_cY_a,$
$X_bY_bX_cX_aY_cY_a, X_bY_bX_cY_cX_aY_a\}.$

To conclude, trace semantics does not preserve refinement for parallel rv-IS like systems.

Scenario equivalence: 1. Dealing with dummy interfaces

Scenario equivalence up to shifting of sub-scenarios with dummy interfaces. In a scenario, the dummy cells, or more generally sub-scenarios with dummy interfaces, may be shifted preserving the scenario structure. As an example, remark that the following scenarios are equivalent

$$v_1 = \begin{array}{ccccc} & * & & * & \\ * & X_b & . & Y_b & * \\ & * & & * & \\ * & X_c & * & Y_c & * \\ & * & & * & \end{array}, \quad v_2 = \begin{array}{ccccc} & * & & * & \\ . & \mathbf{I} & . & Y_b & * \\ & * & & * & \\ * & X_b & . & \mathbf{I} & . \\ & * & & * & \\ * & X_c & * & Y_c & * \\ & * & & * & \end{array}, \text{ and } v_3 = \begin{array}{ccccc} & * & & * & \\ * & X_b & . & \mathbf{I} & . \\ & * & & * & \\ . & \mathbf{I} & . & Y_b & * \\ & * & & * & \\ * & X_c & * & Y_c & * \\ & * & & * & \end{array}$$

Notice that the flattening operator on grids $\flat$ does not take into account the interfaces, in particular dummy interfaces, and it constraints the associated traces to preserve the formal causal top-down and right-left order of the grid cells. In order to get rid of this false causality we apply the following procedure:

1. First, consider all scenarios equivalent (via shifting) to the original one.
2. Next, go to the associated grids with extended letters obtained by inserting the state and class data into the cells.
3. Finally, apply the flattening operator $\flat$ on the resulting grids (over the extended alphabet).

This way, for instance, in the traces associated to v_1 one can find also the trace $\mathbf{I}\,Y_bX_b\,\mathbf{I}\,X_cY_c$ of v_2 with Y_b before X_b which is not directly possible from v_1.

Flattening vs. refinement. Continuing the example in the previous subsubsection, we remark that v was refined as v_1, which is also equivalent with v_2, v_3 and v_4, where

$$v_1 = \begin{array}{ccccc} & * & & * & \\ . & X_b & . & Y_b & . \\ & * & & * & \\ * & X_c & * & Y_c & * \\ & * & & * & \\ . & X_a & . & Y_a & . \\ & * & & * & \end{array}. \quad v_2 = \begin{array}{ccccc} & * & & * & \\ . & \mathbf{I} & . & Y_b & . \\ & * & & * & \\ . & X_b & . & \mathbf{I} & . \\ & * & & * & \\ * & X_c & * & Y_c & * \\ & * & & * & \\ . & X_a & . & Y_a & . \\ & * & & * & \end{array}. \quad v_3 = \begin{array}{ccccc} & * & & * & \\ . & X_b & . & Y_b & . \\ & * & & * & \\ * & X_c & * & Y_c & * \\ & * & & * & \\ . & \mathbf{I} & . & Y_a & . \\ & * & & * & \\ . & X_a & . & \mathbf{I} & . \\ & * & & * & \end{array} \quad v_4 = \begin{array}{ccccc} & * & & * & \\ . & \mathbf{I} & . & Y_b & . \\ & * & & * & \\ . & X_b & . & \mathbf{I} & . \\ & * & & * & \\ * & X_c & * & Y_c & * \\ & * & & * & \\ . & \mathbf{I} & . & Y_a & . \\ & * & & * & \\ . & X_a & . & \mathbf{I} & . \\ & * & & * & \end{array}$$

Even more equivalent scenarios are obtained if X_b, Y_b, X_a, Y_a are split into further sub-actions. Considering the equivalent grids v_2, v_3, v_4 one gets even more traces associated to this refined grid v_1 (i.e., traces with Y_b before X_b or with Y_a before X_a).

To conclude, refinement may work well with grids, even with dummy interfaces, but not with the sets of flattened traces associated to grids.

Scenario equivalence: 2. Structural dependencies

The components of the atomic scenario cell $v = \mathrm{A}\,\underset{\mathrm{t}}{\overset{\mathrm{s}}{\boxed{\mathrm{U}}}}\,\mathrm{B}$ are denoted as follows (using the north / west / east / south / center positions):

```
    s=v.n,  A=v.w,  B=v.e,  t=v.s,  U=v.c.
```

For a scenario w, represented as a grid, we built up an associated X *dependencies graph* $D_X(w)$ representing its cells and their connections as follows:

1. The graph $D_X(w)$ has as nodes the inputs and the outputs of w and the non-constants cells; the latter are the cells with labels not in a given set X of wiring constants. Each node C corresponding to a cell has four connecting ports $C.\alpha$ with $\alpha \in \{w, n, e, s\}$.

2. In $D_X(w)$ there is an edge between $C1.i$ and $C2.j$ if the i-th border of $C1$ is connected with the j-th border of $C2$ by a wire built up from a chain of constants in X.

An example is shown below:

$$w = \begin{array}{ccccc} X & Y & \mathbf{-} & \mathbf{-} & \mathbf{-} \\ \llcorner & + & \urcorner & 0 & 0 \\ \mathbf{-} & + & Z & \urcorner & 0 \\ 0 & \llcorner & + & + & \urcorner \\ 0 & 0 & \mathbf{|} & \llcorner & W \end{array} \; ;$$

$D(w) : (2, 2) \to (2, 2)$ is the graph with 2 inputs on each north and west borders, 2 outputs on the east and south borders, and the edges: $\{(in1.n, X.n), (in2.n, Y.n), (in1.w, X.w), (in2.w, Z.w),$ $(X.e, Y.w), (X.s, Z.n), (Y.s, W.n), (Z.e, W.w),$ $(Y.e, out1.e), (W.e, out2.e), (Z.s, out1.s), (W.s, out2.s)\}$.

The constants used above are: vertical, horizontal, cross, speaker and recorder identities - see the informal presentation below and the semantics formally defined in Section 3. For instance, 4 crosses, a speaker and a recorder connect $Y.s$ with $W.n$ in the example above.

Definition: Let X be a set of constant cells used for wiring. We say two scenarios v, w are X *structurally equivalent* if the associated X dependencies graphs $D_X(v)$ and $D_X(v)$ are isomorphic.

The definition may be instantiated upon the set of specified connections X, particularly for the sets *Connect-1* and *Connect-2* below. One may consider two types of notation for these wiring constants: a textual notation (column 1, below) and a kind of mathematically notation (column 2, below).

The set *Connect-1* consists of:

h	━	*horizontal line* (the message is passing);
v	┃	*vertical line* (the process stays active, doing nothing);
c	$+$	*cross* (the process lets the message pass and is doing nothing);
0	0	*empty cell* (the process is terminated and no message is passing);
s	└	*speaker* (the process passes its state as a message and terminates);
r	┐	*recorder* (a terminated process grabs a message and becomes active starting from the state specified by the message).

The set *Connect-2* is Connect-1 completed with the following "branching connectors":

a	┣	*active speaker* (the process passes its state as a message and remains active);
t	┳	*transparent recorder* (a terminated process sees a message, becomes active starting from the state specified by the message, and lets the message pass);
k	⊥	*terminate a process*;
b	⊣	*block a message*.

In this section we only use the constants in *Connect-1*, hence the prefix X in "X structurally equivalent" relation will be omitted in the forthcoming presentation.

Example 1. A first example of structurally equivalent scenarios is:

$$v = \begin{array}{c} X \\ Y \end{array} \text{ and } w = \begin{array}{cc} X & - \\ \llcorner & \urcorner \\ - & Y \end{array} \text{ (the trace below corresponds to the parssing } \begin{array}{cc} 1 & 2 \\ 3 & 5 \\ 4 & 6 \end{array} \text{ of } w\text{)}$$

Explanation: In v there is only one possible trace XY: a process $p1$ has two transactions to do, first it is doing X, then Y. In the equivalent w scenario, there are two processes and three transactions. One possible trace (out of 42) is $XhshrY$, corresponding to the flattening order described above (next to w). It says that: $p1$ is doing X, while $p2$ is still unborn letting the message to pass; then $p1$ passes its state to $p2$ and it terminates, ignoring the next message; next, $p2$ grabs $p1$'s state and activates itself with that states; finally, $p2$ executes Y, receiving the message from the environment (the one ignored by $p1$).

Example 2. A more complex example of structurally equivalent scenarios is:

$$v = \begin{array}{cc} X & Y \\ Z & W \end{array} \text{ and } w = \begin{array}{ccccc} X & Y & - & - & - \\ \llcorner & + & \urcorner & 0 & 0 \\ - & + & Z & \urcorner & 0 \\ 0 & \llcorner & + & + & \urcorner \\ 0 & 0 & | & \llcorner & W \end{array} \quad \text{(the parssing } \begin{array}{ccccc} 1 & 3 & 6 & 9 & 15 \\ 2 & 4 & 8 & 14 & 22 \\ 5 & 7 & 10 & 18 & 23 \\ 11 & 12 & 17 & 19 & 24 \\ 13 & 16 & 20 & 21 & 25 \end{array} \text{ is used in the}$$

trace below)

Explanation: In v there are two processes and two transactions. A possible trace for v (out of 2) is $XYZW$ saying that: $p1$ is doing X and passes message to $p2$ which is doing Y; then $p1$ is doing Z and passes message to $p2$ which is doing W. On the other hand, there are a huge number of traces in w (i.e., 701149020), one of them being $XsYchhcrhZ0s00h0crcvs00rW$, corresponding to the order shown next to w above. This describes a complex execution of the same basic tasks X, Y, Z, W, now using 5 processes and 5 transactions: process $p1$ is doing X and passes message to $p2$; then it sends its state to $p2$ and terminates; $p2$ is doing Y and passes a message to $p3$; then $p2$ does nothing, ignoring the message from $p1$, as well; $p1$ ignores the message from the environment, and the same is doing $p3$ with the message of $p2$; then $p2$ does nothing, ignoring the message from $p1$, as well; $p3$ grabs the message of $p1$ and it activates itself with the state of $p1$; next, $p4$ ignores the message from $p3$; then, $p3$ is performing task Z; and so on. The difficulty in finding the explanation for the chosen trace execution of the above scenario is obvious, while an explanation for the whole 701149020 trace-based executions is probably completely meaningless. On the other hand, the meaning of activities specified by the scenario w is pretty much clear from its given two-dimensional specification. To conclude, traces are not appropriate for representing the runs in parallel rv-IS like computing systems.

Example 3: Modeling "back arrows". In the examples to follow we will often use a shortcut notation: *scenarios with back-arrows*. An example is presented in Fig. 2.4(a).

The plain scenarios for modeling scenarios with "back arrows" are easily described as in Fig. 2.4(c). Plain scenarios associated to scenarios with back arrows are a particular case of the kind of scenarios presented in this subsection. This allows us to extend the structural equivalence definition to scenarios with back-arrows.

Scenario equivalence: 3. Structural extension

A next discussion on scenario comparison is related to a kind of “structural extension” relation, i.e., with the possibility to have more internal causality structure in a refined model.

We say a scenario w X *structurally extends* a scenario v if the dependencies graph $D_X(v)$ of v is a subgraph of $D_X(w)$ and, moreover, $D_X(w)$ has the same nodes and the same inputs and outputs as $D_X(v)$. In other words, while having the same non-constants cells (i.e., the cells not in X are the same) and the same inputs and outputs, the refined scenario w may have more internal connections than v.

For an example, consider the scenarios

$$v = \begin{array}{c} *\ \ * \\ .X_b.Y_b. \\ *\ \ * \\ *X_c*Y_c* \\ *\ \ * \\ .X_a.Y_a. \\ *\ \ * \end{array} \text{ and } w = \begin{array}{c} *\ \ * \\ .X_b*Y_b. \\ *\ \ * \\ *X_c*Y_c* \\ *\ \ * \\ .X_a.Y_a. \\ *\ \ * \end{array}$$

Then, w is a structural extension of v having an additional not-dummy connection between X_b and Y_b.

Scenario equivalence: 3. Sub-scenario stuttering

Finally, we consider sub-scenarios stuttering. This allows to insert or remove sub-scenarios of a scenario, provided the connecting state and class interfaces with the remaining scenario does not change.

Let v be a scenario and c a "whole" in v, i.e., a non-selfintersecting path via neighboring states and classes with an empty contents. Let w be a scenario with the same interface as c. If we replace in v the whole c by w we get a new scenario $v\{w/c\}$ considered to be in an *1-step stuttering relation* with v, denoted $v =_{1St} v\{w/c\}$. The *scenario stuttering equivalence* $=_{St}$ is the equivalence relation generated by the 1-step stuttering relation $=_{1St}$.

Actually, the stuttering equivalence $=_{St}$ can be obtained as the symmetric, reflexive, and transitive closure of $=_{1St}$.

As an example, we consider the scenarios v and w described in the pictures below (for the sake of simplicity, the data around the cells are omitted)

$$v = \begin{array}{cccc} \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{d} \\ \mathtt{e} & \mathtt{f} & \multicolumn{1}{|c}{\mathtt{g}} & \mathtt{h} \\ \cline{2-3} \mathtt{i} & \mathtt{j} & \multicolumn{1}{|c}{\mathtt{k}} & \mathtt{l} \\ \mathtt{m} & \mathtt{n} & \mathtt{o} & \mathtt{p} \end{array} \text{ and } w = \begin{array}{ccccc} \mathtt{a} & \mathtt{b} & . & \mathtt{c} & \mathtt{d} \\ \cline{3-3} \mathtt{e} & \mathtt{f} & \multicolumn{1}{|c|}{\mathtt{X}} & \mathtt{g} & \mathtt{h} \\ \cline{2-2} \cline{4-4} . & \multicolumn{1}{|c}{\mathtt{Y}} & \mathtt{Z} & \multicolumn{1}{c|}{\mathtt{U}} & . \\ \cline{2-2} \cline{4-4} \mathtt{i} & \mathtt{j} & \multicolumn{1}{|c|}{\mathtt{V}} & \mathtt{k} & \mathtt{l} \\ \cline{3-3} \mathtt{m} & \mathtt{n} & . & \mathtt{o} & \mathtt{p} \end{array} .$$

The "whole" in v consists of a circular line corresponding to the lines inserted in the picture; one possible description is (starting from the center): up-down-left-right-down-up-right-left. Suppose the cross represented by `X,Y,Z,U,V` in w is such that:

1. `X,V` have the data on the eastern borders equal to that of the western borders and, moreover, the northern interface of `X` and the southern interface of `V` are `nil`; and

2. `Y,U` have the data on the southern borders equal to that of the northern borders and the western interface of `Y` and the eastern interface of `U` are `nil`.

Then, w and v are in the 1-step stuttering relation, hence they are stuttering equivalent.

2.1.5 Refinement of register-voice interactive systems

The rv-IS model is a combination of state-based computing systems and interactive dataflow-like computing systems. An appropriate rv-IS refinement definition has to obey the following constraints:

(1) By restriction to systems with no interaction classes it should reduce itself to the usual refinement of classical state-based systems (a presentation of this type of refinement may be found in [3]);

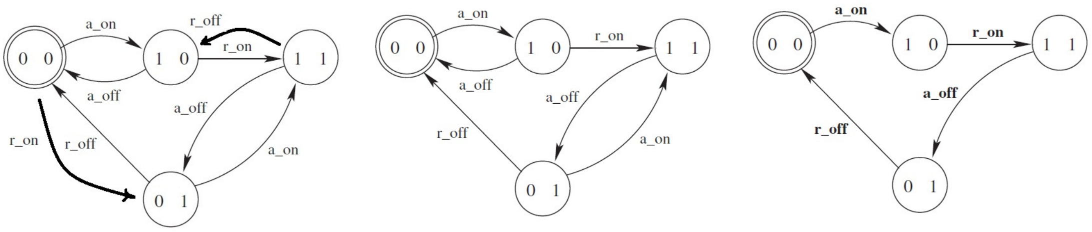
(2) Similarly, by restriction to systems with no states it has to produce a refinement for interactive dataflow networks (see [26] for a study of this type of refinement).

For usual sequential computing systems there are several approaches to define refinement relations. A simple approach is to use traces: in terms of traces, except for some additional technical conditions, a concrete systems C is a refinement of an abstract one A if the traces of C represent a subset of the traces of A, modulo a relation connecting the state of C to those of A. A more efficient method is to define the refinement relation using the syntactical representation of the systems. This latter approach avoid the burden of computing the traces associated to a system, while ensuring the validity of the inclusion relation between the associated traces required by the previous definition.

In this subsection we present a notion of refinement of rv-IS systems in terms of associated scenarios. Scenarios are an extension of traces, hence this approach directly lifts the first refinement definition above to the level of rv-IS systems. It is possible to define a refinement relation using the syntactical representation of unstructured rv-IS systems, but this requires a detailed presentation of unstructured rv-IS systems and of their algebraic representation and it is not included in this section.

Refinement in state-based computing systems

Refinement of classical sequential specifications/programs has a long history, see [1, 42, 26, 30, 23, 3, 38, 39, 16, 18], to mention just a few pointers to this active research topics. We sketch here the key features of the approach following Abrial's book [3].



(a) Initial automaton A1

(b) 1st refined automaton A2

(b) 2nd refined automaton A3

Figure 2.6: Automata for an "action-reaction" pattern

As a running example, we use nondeterministic finite automata and Event-B models for an "action-reaction" system. Automata models of these systems are presented in Fig. 2.6.

Refinement between such systems may be easily defined using traces. We say a system C is a *refinement* of a system A if the following conditions holds:

(TI) *trace inclusion*

(SI) *stuttering invariance*

(RDF) *relative deadlock freedom.*

The explanation of the conditions' meaning is below.

The basic constraint on refinement is (TI), stating that the traces of the refined model C are a subset of the traces of the abstract model A. This property is the corner stone of the refinement method producing a strategy to develop correct-by-construction implementations. The approach starts with a general, abstract, and often nondeterministic specification. Gradually, refined models with less degree of nondeterminism are produced till the very end when hopefully a deterministic model is obtained. Moreover, the final model is supposed to be close to the execution system, getting a straightforward implementation.

This straightforward relation has to be extended to cope with data refinement. To this end, trace equality is to be considered up to a *stuttering relation* (i.e., state repetition in the traces); this is condition (SI). One simple example is when a distinction makes sense between "internal" and "external" variables. The traces in the concrete model C may have many details which are presented in terms of internal variables, while in A one can see only the external variables. During this projection from C to A a sequence of states in a C trace may have no visible changes in terms of external variables, hence stuttering states in the associated A trace may occur.

Technically, one more condition is usually added: (RDF). Its role is to avoid having deadlock in a state of C corresponding to a state of A with no deadlock, at all. Without (RDF) the empty model will refine any model. While an exact capture of this property requires the computation of the transitive-reflexive closure of the abstract and concrete transition relations ae/re of the systems A/C, there is a simpler, but slightly more restrictive condition "first step relative deadlock freedom" (FSRDF). The mathematical formulation of (FSRDF) is shown as the last condition in Fig. 2.7(b).

$$
\begin{aligned}
L &= \{0 \mapsto 0\} \\
ae &= \{\ (0 \mapsto 0) \mapsto (1 \mapsto 0), \\
&\qquad (0 \mapsto 1) \mapsto (1 \mapsto 1), \\
&\qquad (1 \mapsto 0) \mapsto (0 \mapsto 0), \\
&\qquad (1 \mapsto 1) \mapsto (0 \mapsto 1), \\
&\qquad (1 \mapsto 0) \mapsto (1 \mapsto 1), \\
&\qquad (0 \mapsto 1) \mapsto (0 \mapsto 0)\,\}
\end{aligned}
\qquad
\begin{aligned}
M &= \{0 \mapsto 0\} \\
re &= \{\ (0 \mapsto 0) \mapsto (1 \mapsto 0), \\
&\qquad (1 \mapsto 1) \mapsto (0 \mapsto 1), \\
&\qquad (1 \mapsto 0) \mapsto (1 \mapsto 1), \\
&\qquad (0 \mapsto 1) \mapsto (0 \mapsto 0)\,\}
\end{aligned}
$$

(a) Transition relations ae and re

$$
\boxed{
\begin{array}{l}
M \subseteq L \\[1ex]
M \neq \varnothing \\[1ex]
re \subseteq ae \\[1ex]
\mathrm{dom}(ae) \subseteq \mathrm{dom}(re)
\end{array}
}
$$

(b) Mathematical definition

Figure 2.7: Refinement definition

The conditions (TI), (SI), and (RDF) are easily verified on examples A1/A2 and A2/A3 in Fig. 2.6, hence we have a chain of refinement in that figure.

Refinement in interaction-based systems

For the interaction part, we intend to follow Broy's approach [26] to define refinement in dataflow-based interactive systems.

The key features of Broy's approach are:

1. The study is based on streams and an algebra DFN of dataflow networks (see [28]).
2. Refinement is compatible with the algebra operations. This means, if the arguments are refined systems, then the results of the DFN operations are also refined systems.
3. The number of the channels may be different in the abstract and the refined model. This is a technique to model "new" or "hidden" channels/events.

4. Data-refinement is allowed to act at the level of stream values.

The full integration of these ideas into the rv-IS formalism requires the development of an algebra for rv-IS systems and it is not in the scope of this paper. To achieve the desiderate of capturing Broy's method, the rv-IS algebra has to extend the DFN algebra.

Refinement in state- and interaction-based systems

Why should one study rv-IS models and not only separate state-based and interaction-based models? A few advantages may be:

- The rv-IS approach gives a better (structured, compositional) way to handle shared events or shared variables in classical systems using a dataflow-like interaction model;
- It increases the expressivity power of dataflow-like interaction systems by including complex, structural, state-based control mechanisms.

It is not hard to development an "algebra" for representing such rv-IS systems, similar to the network algebra approach presented in [53, 28]. We hope the notion of refinement to be introduce below is compatible with this algebra, hence to get compositional refinement in rv-IS systems.

From the refinement definition of classical computing systems, we will take a closer look to the conditions (TI) and (SI) trying to lift them to the rv-IS scenario level. The last condition (RDF) it is somehow problematic. In its strong form, using transitive-reflexive closure, the condition may be usefulness: indeed, while in the case of finite automata the reachability property if decidable, for finite interactive systems this property is not decidable [51]. However, (RDF) can be introduced for systems where the behaviors on states and on classes are independent: just write separately the condition for states and for classes. Anyways, the stronger (FSRDF) condition can be easily stated.

Refinement definition. The definition of refinement of rv-IS models below is a natural lifting of the trace-based definition of refinement of automata and of Event-B models. Stuttering equivalence on traces is replaced by the scenario equivalence introduced in the previous subsection.

Definition: Given two rv-IS systems AIS and CIS, we say CIS is a *refinement* of AIS if the following conditions hold:

- The scenarios of CIS are a *subset* of the scenarios of AIS, under the following assumptions:

1. The scenario equality is up to the *structural extension* and *stuttering equivalence* relations defined in the previous subsection.
2. The scenarios are *projected on states and classes*, hence the cells' labels does not matter.
3. For comparison one uses a *gluing correspondence* between the state and the class variables of the CIS and the AIS systems.

- A scenario in CIS corresponding to as abstract scenario in AIS *can be extended* in CIS whenever the abstract one can be extended in AIS.

Often, the states and classes of the AIS system are also present in the CIS system, hence the correspondence mentioned in the definition is a simple projection: in that case, ignoring the new state and class variables used in the CIS scenarios one gets scenarios written in the AIS variables.

Examples: Comparing Event-B and rv-IS refinements We presents a few simple examples to get intuition on rv-IS refinement. They use the "action-reaction" model introduced in this subsection earlier.

First rv-IS model for A/R-pattern: No interaction. We start with the Event-B model $M1$ given by the following events (its possible traces are those generated by the automata $A1$ from Fig. 2.6):

$M1$ ::

```
init        a_on        a_off       r_on        r_off
  a := 0      when        when        when        when
  r := 0        a = 0       a = 0       r = 0       r = 1
              then        then        then        then
                a := 1      a := 0      r := 1      r := 0
              end         end         end         end
```

An associated rv-IS model $IS1$ is specified as follows:

- the initial nodes are X, nil, s; the final ones are Z, nil, a_0, r_0;
- the transactions are:

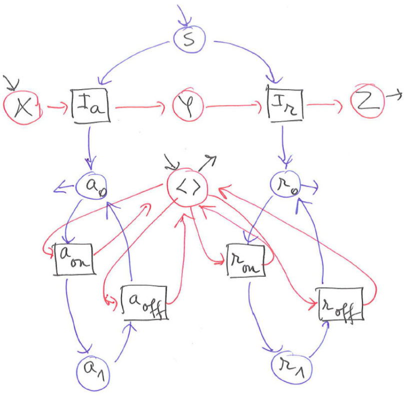
$$\begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & s & \multicolumn{1}{c}{} \\ \hline X & I_a & Y \\ \hline \multicolumn{1}{c|}{} & a_0 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array} \quad \begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & s & \multicolumn{1}{c}{} \\ \hline Y & I_r & Z \\ \hline \multicolumn{1}{c|}{} & r_0 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array} \quad \begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & a_0 & \multicolumn{1}{c}{} \\ \hline nil & a_{on} & nil \\ \hline \multicolumn{1}{c|}{} & a_1 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array} \quad \begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & a_1 & \multicolumn{1}{c}{} \\ \hline nil & a_{off} & nil \\ \hline \multicolumn{1}{c|}{} & a_0 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array}$$

$$\begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & r_0 & \multicolumn{1}{c}{} \\ \hline nil & r_{on} & nil \\ \hline \multicolumn{1}{c|}{} & r_1 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array} \quad \begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & r_1 & \multicolumn{1}{c}{} \\ \hline nil & r_{off} & nil \\ \hline \multicolumn{1}{c|}{} & r_0 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array}$$

A graphical representation of $IS1$ is shown in Fig. 2.8, where the nil class is represented by a dot '.' (or '$\langle\rangle$').

This rv-IS model $IS1$ is with independent actions and it corresponds to $A1$. Two scenarios are shown in Fig. 2.8(b). In the second scenario, the data around the cells are omitted. In this model the a and r columns can slide as there are no interactions between the associated actions. It is not hard to see that if one restricts to $*_{on}$ and $*_{off}$ actions, then precisely the traces of automaton $A1$ are obtained.

Actually, these are action-traces resulting by the projection on the center cells of the traces obtained form the scenarios in the way described in the previous subsection via grids over extended alphabets and the flattening operator. As we already said, from action-traces one can get state-and-class traces in a straightforward way. So, in this and the forthcoming examples of this subsections, traces means action-traces and are used for automata, Event-B and rv-IS models.



(a) Graphical representation of IS1

	s		s	
X	I_a	Y	I_r	Z
	a_0		r_0	
·	I	·	r_on	·
	a_0		r_1	
·	a_on	·	I	·
	a_1		r_1	
·	I	·	I	·
	a_1		r_1	
·	a_off	·	I	·
	a_0		r_1	
·	I	·	r_off	·
	a_0		r_0	

	*		*	
*	I_a	*	I_r	*
	*		*	
·	I	·	r_on	·
	*		*	
·	a_on	·	I	·
	*		*	
·	I	·	r_off	·
	*		*	
·	a_off	·	r_on	·
	*		*	
·	I	·	r_off	·
	*		*	

(b) Two scenarios

Figure 2.8: First Action/Reaction rv-IS model (independent actors)

Relating IS and Event-B models: First case. If one ignore *Connect-1* actions, then the traces associated to the scenarios of the rv-IS model $IS1$ are the same as the traces of the Event-B $M1$; moreover, ignoring the initialization actions Ia, Ir, these traces are the traces recognized by automaton $A1$.

Second rv-IS model for A/R-pattern: One-way interaction. This second example $IS2$ is a refinement of the former and it corresponds to the automaton in Fig. 2.6(b). In this refined model "r sees a" (it corresponds to automaton $A2$). A graphical representation of $IS2$ is shown in Fig. 2.9, together with a typical scenario.

The Event-B model $M2$ is specified by the events

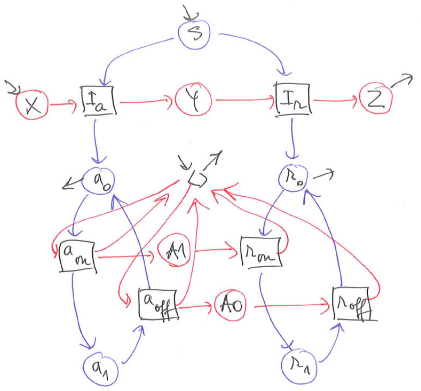
$M2 ::$	init	a_on	a_off	r_on	r_off
	$a := 0$ $r := 0$	**when** $a = 0$ **then** $a := 1$ **end**	**when** $a = 0$ **then** $a := 0$ **end**	**when** $\underline{a = 1}$ $r = 0$ **then** $r := 1$ **end**	**when** $\underline{a = 0}$ $r = 1$ **then** $r := 0$ **end**

while the corresponding rv-IS model $IS2$ is specified as follows:

- X, nil, s are initial nodes; Z, nil, a_0, r_0 are final nodes;
- the transactions are:

$$\begin{array}{|c|c|c|} \cline{2-2} & s & \\ \hline X & I_a & Y \\ \hline & a_0 & \\ \cline{2-2} \end{array} \quad \begin{array}{|c|c|c|} \cline{2-2} & s & \\ \hline Y & I_r & Z \\ \hline & r_0 & \\ \cline{2-2} \end{array} \quad \begin{array}{|c|c|c|} \cline{2-2} & a_0 & \\ \hline nil & a_{on} & nil \\ \hline & a_1 & \\ \cline{2-2} \end{array} \quad \begin{array}{|c|c|c|} \cline{2-2} & a_1 & \\ \hline nil & a_{off} & nil \\ \hline & a_0 & \\ \cline{2-2} \end{array}$$

$$\begin{array}{|c|c|c|} \cline{2-2} & a_0 & \\ \hline nil & a_{on} & A1 \\ \hline & a_1 & \\ \cline{2-2} \end{array} \quad \begin{array}{|c|c|c|} \cline{2-2} & a_1 & \\ \hline nil & a_{off} & A0 \\ \hline & a_0 & \\ \cline{2-2} \end{array} \quad \begin{array}{|c|c|c|} \cline{2-2} & r_0 & \\ \hline A1 & r_{on} & nil \\ \hline & r_1 & \\ \cline{2-2} \end{array} \quad \begin{array}{|c|c|c|} \cline{2-2} & r_1 & \\ \hline A0 & r_{off} & nil \\ \hline & r_0 & \\ \cline{2-2} \end{array}$$



(a) Graphical representation of IS2

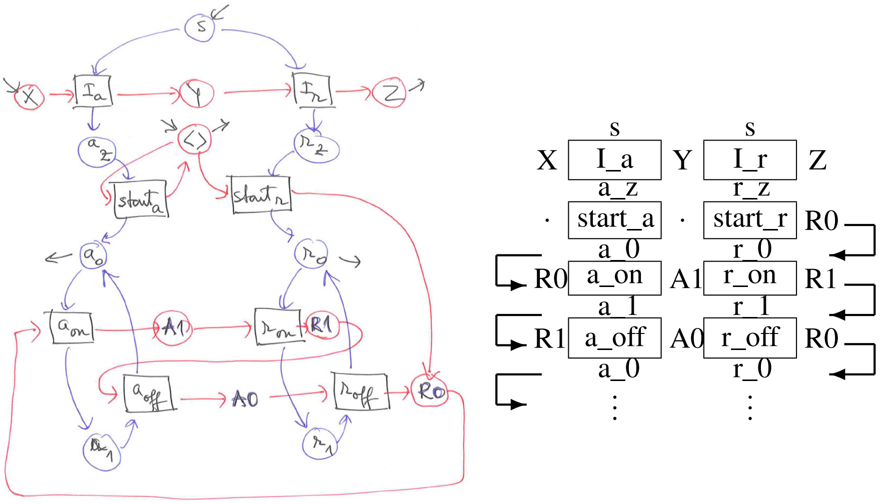
	s		s	
X	I_a	Y	I_r	Z
	a_0		r_0	
·	a_on	A1	r_on	·
	a_1		r_1	
·	a_off	·	I	·
	a_0		r_0	
·	a_on	·	I	·
	a_0		r_0	
·	a_off	A0	r_off	·
	a_0		r_0	
	⋮		⋮	

(b) A scenario

Figure 2.9: Second Action/Reaction rv-IS model (one-way interacting actors)

Relating IS and Event-B models: Second case. If one ignore the initialization actions Ia, Ir and the *Connect-1* actions, then the traces of $IS2$ are the same as the traces of $M2$ (and are those of automaton $A2$).

The "one-way" interaction constraint produces simple scenarios. Actually, the scenarios in $IS2$ structural extend scenarios in $IS1$. Indeed, the only difference is that connections between `a_on/a_off` and `r_on/r_off` replace some dummy connections from $IS1$ scenarios. This shows that, according to rv-IS refinement definition, $IS2$ is a refinement of $IS1$.



(a) Graphical representation of IS3 (b) A scenario

Figure 2.10: Third Action/Reaction rv-IS model (cyclic interacting actors)

Third rv-IS model for A/R-pattern: Cyclic interaction. This final refinement corresponds to the third automaton $A3$ in Fig. 2.6(c). The event-B model is

$M3$ ::

init	a_on	a_off	r_on	r_off
$a := 0$ $r := 0$	**when** $a = 0$ $r = 0$ **then** $a := 1$ **end**	**when** $a = 0$ $r = 1$ **then** $a := 0$ **end**	**when** $a = 1$ $r = 0$ **then** $r := 1$ **end**	**when** $a = 0$ $r = 1$ **then** $r := 0$ **end**

and the corresponding rv-IS model $IS3$ is defined by:

- the initial classes and states are X, nil, s; the final classes and states are Z, nil, a_0, r_0;
- the transactions are:

$$\begin{array}{|c|c|c|}\hline & s & \\ \hline X & I_a & Y \\ \hline & a_z & \\ \hline\end{array}\quad\begin{array}{|c|c|c|}\hline & s & \\ \hline Y & I_r & Z \\ \hline & r_z & \\ \hline\end{array}\quad\begin{array}{|c|c|c|}\hline & a_z & \\ \hline nil & start_a & nil \\ \hline & a_0 & \\ \hline\end{array}\quad\begin{array}{|c|c|c|}\hline & r_z & \\ \hline nil & start_r & R0 \\ \hline & r_0 & \\ \hline\end{array}$$

$$\begin{array}{|c|c|c|}\hline & a_0 & \\ \hline R0 & a_{on} & A1 \\ \hline & a_1 & \\ \hline\end{array}\quad\begin{array}{|c|c|c|}\hline & a_1 & \\ \hline R1 & a_{off} & A0 \\ \hline & a_0 & \\ \hline\end{array}\quad\begin{array}{|c|c|c|}\hline & r_0 & \\ \hline A1 & r_{on} & R1 \\ \hline & r_1 & \\ \hline\end{array}\quad\begin{array}{|c|c|c|}\hline & r_1 & \\ \hline A0 & r_{off} & R0 \\ \hline & r_0 & \\ \hline\end{array}$$

In this final model both “r sees a” and “a sees r” (it corresponds to $A3$). Traces include:

$$I_a I_r \dots a_{on} r_{on} \dots a_{\mathit{off}} r_{\mathit{off}} \dots a_{on} r_{on} \dots a_{\mathit{off}} r_{\mathit{off}},$$

where '...' represents hidden sequences of *Connect-1* actions of $IS3$ scenarios, in particular those actions used for modeling the ï£¡back-arrowsï£¡.

A similar result holds: (1) If one abstracts from *Connect-1* invisible actions, the traces associated to $IS3$ scenarios are the same with those of $M3$ (and are the traces of $A3$). (2) $IS3$ is a refinement of $IS2$.

2.1.6 From Event-B to structured rv-IS

In this subsection we will propose a general method to transform an Event-B system specification into an rv-IS specification. The method actually produces an rv-program, whenever the transformations used to define events' actions can be implemented with a code written in the rv-module code syntax. As already mentioned at the beginning of this section, we can also comment on the correctness of the EB2IS translation. One possibility to do this is via the trace semantics. For an Event-B model M, one can define the traces associated to all possible runnings of the model. On the other hand, one can consider the translated rv-IS model M' and the associated running scenarios; starting with these scenarios and using the flattening operator, a set of traces may be associated to the translated model M'. The correctness problem for the EB2IS translation must show that the translation preserves the associated traces, up to state stuttering and a hiding of the auxiliary variables and modules introduced by the translation. Without giving a formal proof that the translation is correct according to this definition, the examples included in this subsection 2.1.6 provide evidence in this direction.

For this moment, we will deal only with the events of a certain system, not with the invariants of that system.

Usually, a system in Event-B can be seen as a set of events which have the following form:

```
Event-i
 when
  Gi
 then
  Ai
 end
```

where for each `i`, `Event-i` is the name of the event, `Gi` is the guard and `Ai` is the action. Notice that usually, `Gi` and `Ai` are sets of predicates, respectively, actions. A system in Event-B has also a special event, namelly the `Init` event.

Constants, axioms, variables, and invariants may be used for proofs in a similar way in Event-B and rv-IS models.

The rv-IS specification associated to an Event-B model captures not only the Event-B actions, but also the semantics rules used for its execution.

In order to construct a structured rv-IS specification for an Event-B model, we will define a *manager* that will decide which event can take place at the each time. For each event `Event-i`, we construct two modules `Gi` and `Ei` - modules `Gi` are used by the manager in order to decide which event to be triggered, while modules `Ei` are used by the manager to simulate the changes on the system caused by the event. Since there is no possible confusion, we will denote with `Ainit` the actions from the event `Init` in Event-B, with `Gj` the guard of the event `Event-j` and with `Aj` the actions of the event `Event-j` in Event-B.

Since in Event-B, the memory is shared by all events, in the associated rv-IS specification we have to simulate the common memory. Therefore, after each action, the manager must update the variables of all processes.

Let us suppose that the system in Event-B has `N` events, excepting the `Init` event. We define a set $\mathtt{Ev} = \{\mathtt{Ei} \mid \mathtt{i} = \overline{\mathtt{1,N}}\}$ and a set `V` which contains all the constants and all the variables in the system.

```
module I
{listen nil}{read C}{
  Ainit;
  tV = V ∪ C;
}{speak tV}{write nil}
```

```
module ID
{listen tV}{read nil}{
  V = tV;
}{speak tV}{write V}
```

```
module Mg
{listen tV}{read nil}{
  ten = ∅;
}{speak ten}{write nil}
```

```
module Gj
{listen ten}{read V}{
  if(Gj){ten = ten ∪
  {Ej};};
}{speak ten}{write V}
```

```
module Me
{listen ten}{read nil}{
  tk :∈ ten;
  tV = ∅;
}{speak tk, tV}{write
nil}
```

```
module Ej
{listen tk,tV}{read V}{
  if(tk = Ej){Aj; tV = V;}
}{speak tk,tV}{write V}
```

```
module Mu
{listen tk,tV}{read nil}{
  null;
}{speak tV}{write nil}
```

```
module U
{listen tV}{read V}{
  V = tV;
}{speak tV}{write V}
```

Table 2.3: Modules for EB2IS translation

The general format of the corresponding rv-IS specification is the following:

```
1:      (I # for_s(j=1,N){ID})
2:  $   (Mg # for_s(j=1,N){Gj})
3:  $  while_st(ten ≠ ∅) {
4:          (Me # for_s(j=1,N){Ej})
5:        $ (Mu # for_s(j=1,N){U}))
6:        $ (Mg # for_s(j=1,N){Gj})
7:     }
```

Module I contains all the initializations from the event Init in Event-B and the module ID is used in order to provide the same variables to all the processes involved in the program. The manager uses the modules Mg, Me and Mu to simulate the behavior in Event-B. The manager decides which event can take place. In line 2, the manager construct a set ten of possible events that can take place by checking the guards of each event; the module Gj contains the guard of event Event-j. While we have at least one event that can take place, we start to simulate its behavior. In line 4, the manager chooses one event from the list of possible events that can occur at the current moment and starts to search the ‘chosen one’. Module Ej checks if the event Event-j is the ‘chosen one’ and if yes, it makes the modifications in the system w.r.t. to actions in event Event-j by using the set Aj. Finally, in line 5, the manager updates the variables in all processes w.r.t. the new modifications. After this point, we repeat the process until we have no more events that can occur.

We can imagine the behavior of the manager splited in the following actions: search for the ‘chosen’ event (lines 2 and 6), modify the system w.r.t. the actions of the ‘chosen’ event (line 4), update the variables of all processes (line 5). In order to make the next action, the manager needs the information from the previous action, therefore we must compose the lines of model diagonally.

Before presenting the general framework of the modules, let us denote by C the set of all constants and by V the set of all variables of the system in Event-B.

The modules of the associated structured rv-IS specifications are described in Table 2.3.

In this translation the manager decides in an nondeterministic fashion which event can take place. However, in module Me the manager can implement any method for deciding which event can be triggered. The manager described above chooses at any moment one single event that can take place (tk :∈ ten; tk is a single token). In a more general implementation, the manager is free to choose a set of events that can take place at a certain moment of time by constructing tk to be a set, but in this case we have to pay careful attention to avoid written conflicts for updated variables occurring in more than one event.

A general scenario for the program above is presented in Figure 2.11.

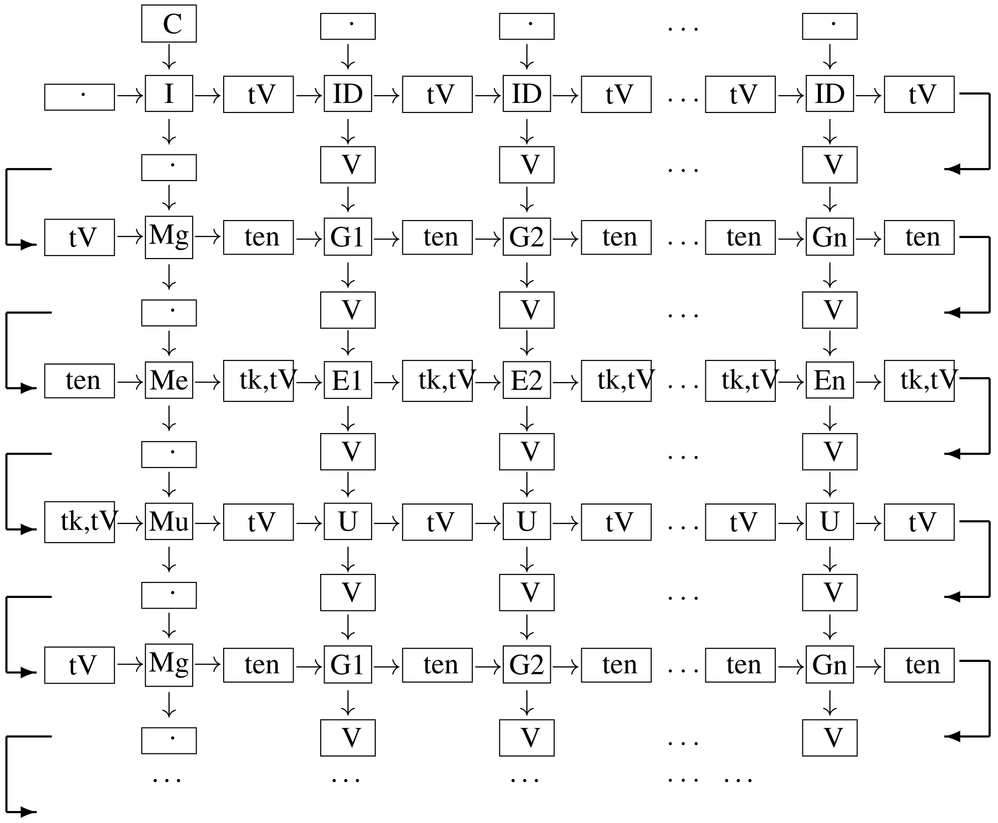


Figure 2.11: Scenarios for an rv-IS obtained from an Event-B model

2.1.7 An example - a simple file transfer protocol

In this subsection we present an example: we take a system modeled in Event-B and translate it into an rv-IS specification. The model in Event-B can be found in [3].

The model is that of a classical file transfer protocol. We want to transfer a sequential file, i.e. a file composed of a finite number of items arranged in a specific order, from one agent, the sender, to another one, the receiver. The transferred file should be equal to the original one. The protocol is a distributed program and it is realized by two distinct programs which exchange various kinds of messages and work on different machines.

We are going to develop the protocol in more steps, taking into account the following refinement strategy. In the initial model, we are interested only in the final result of the protocol, not on how it is achieved. The file in this model is transmitted in one shot. It is important to understand that in this initial model, the agents are not supposed to reside on different sites.

In the first refinement, we start to transmit the file piece by piece between the two agents. The main difference with the initial model is that we separate the sender and the receiver. However, they are not still completely independent, since the receiver can "see" what remains to be transmitted by the sender and it can take "directly" the next item from the sender's file (it can access the sender's memory).

In the next refinement step, the sender and the receiver are completely independent one from another, the receiver has no longer access to the sender's memory. In this stage, the two agents communicate only through messages: the sender is sending messages that are read by the receiver and the receiver responds to these messages by returning an acknowledgment message to the sender. The distributed nature of the protocol is revealed by this refinement step.

Initial model

Event-B model: FTP, initial model In all the models of this file transfer protocol, we assume that we have a nonempty set D (the carrier set), two constants, n, a positive number, and f, a total function from the interval $\{1, \ldots, n\}$ into the set D.

Informally, f is the file to be transferred from the sender's site to the receiver's site. The constant n represents the length of the file f, while D contains the data that can be stored in the file f. Consequently, it is natural to represent the file f as a total function with elements in D.

The result of the protocol is a variable g, the file transferred to the receiver. Since we construct g step by step, we consider that g is a partial function from the interval $\{1, \ldots, n\}$ into the set D.

In the initial model, we say nothing about the internal structure of the file f. In order to transfer this file, we have an event `receive` which choose randomly a partial function g with values in D, until this function is equal to f. When we obtain such a function g, then we can assume that the file f was send to the receiver's site.

The axioms and the invariants are the same and are not explicitly shown in the associated rv-IS models.

The Event-B events of this initial model are the followings:

FTP-EB1 ::=

```
init
  $g :\in \mathbb{N} \leftrightarrow D$
```

```
receive
  when
    $g \neq f$
  then
    $g :\in \mathbb{N} \leftrightarrow D$
  end
```

```
final
  when
    $g = f$
  then
    $skip$
  end
```

Translation of the Event-B model: FTP, initial model In order to construct an rv-IS specification FTP-IS1, let us consider the following set of events $\texttt{Ev} = \{\texttt{Erecv}, \texttt{Efin}\}$.

```
module I1
{listen nil}{read f,n}{
  g :∈ ℕ ↔ D;
  tV = {f,n,g};
}{speak tV}{write nil}
```

```
module ID
{listen tV}{read nil}{
  V = tV;
}{speak tV}{write V}
```

```
module Mg
{listen tV}{read nil}{
  ten = ∅;
}{speak ten}{write nil}
```

```
module Me
{listen ten}{read nil}{
  tk :∈ ten; tV = ∅;
}{speak tk,tV}{write nil}
```

```
module Mu
{listen tk,tV}{read nil}{

}{speak tV}{write nil}
```

```
module U
{listen tV}{read V}{
  V = tV;
}{speak tV}{write V}
```

```
module Grecv1
{listen ten}{read V}{
  if(g ≠ f){
   ten = ten U {Erecv};};
}{speak ten}{write V}
```

```
module Gfin1
{listen ten}{read V}{
  if(g = f){
   ten=ten U {Efin};};
}{speak ten}{write V}
```

```
module Erecv1
{listen tk,tV}{read V}{
  if(tk = Erecv){
   g :∈ ℕ ↔D; tV = V;};
}{speak tk,tV}{write V}
```

```
module Efin1
{listen tk,tV}{read V}{
  if(tk = Efin){tV = V;};
}{speak tk,tV}{write V}
```

Table 2.4: Modules for FTP-IS1 specification

The specification is:

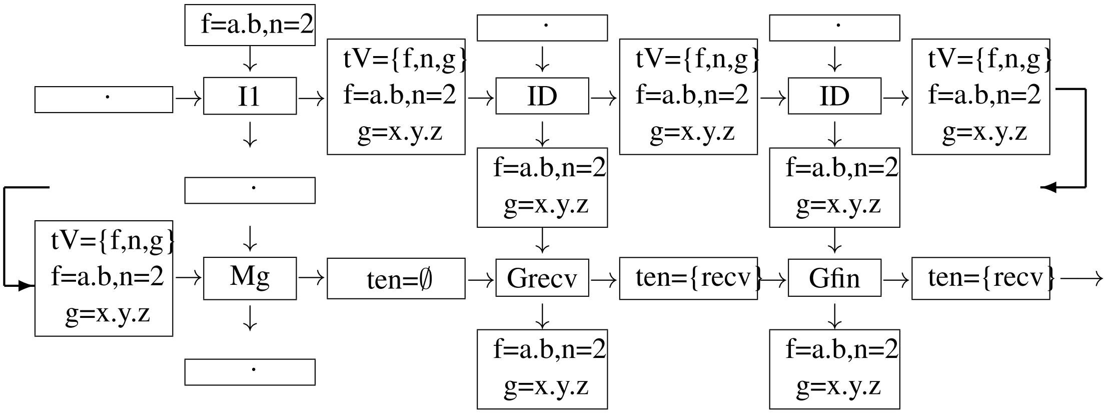
```
FTP-IS1 ::=    (I1 # ID # ID)
             $ (Mg # Grecv1 # Gfin1)
             $ while_st(ten ≠ ∅) {
                    (Me # Erecv1 # Efin1)
                  $ (Mu # U # U)
                  $ (Mg # Grecv1 # Gfin1)
               }
```

where the involved modules `I, ID, Mg, Me, Mu, Grecv, Erecv, Gfin, Efin, U` are those described in Table 2.4.

Let us analyze a simple case: suppose that `f` contains only two characters, say `f=a.b`; thus `n=2`. A typical scenario for the FTP-IS1 specification is built up using the following partial scenarios. In the presentation, `g=x.y.z, g=s.t, ..., g=a.b` is just a sequence of random assignments for `g`. Alternatively, one can consider the case where the lucky assignment `g=a.b` never occurs.

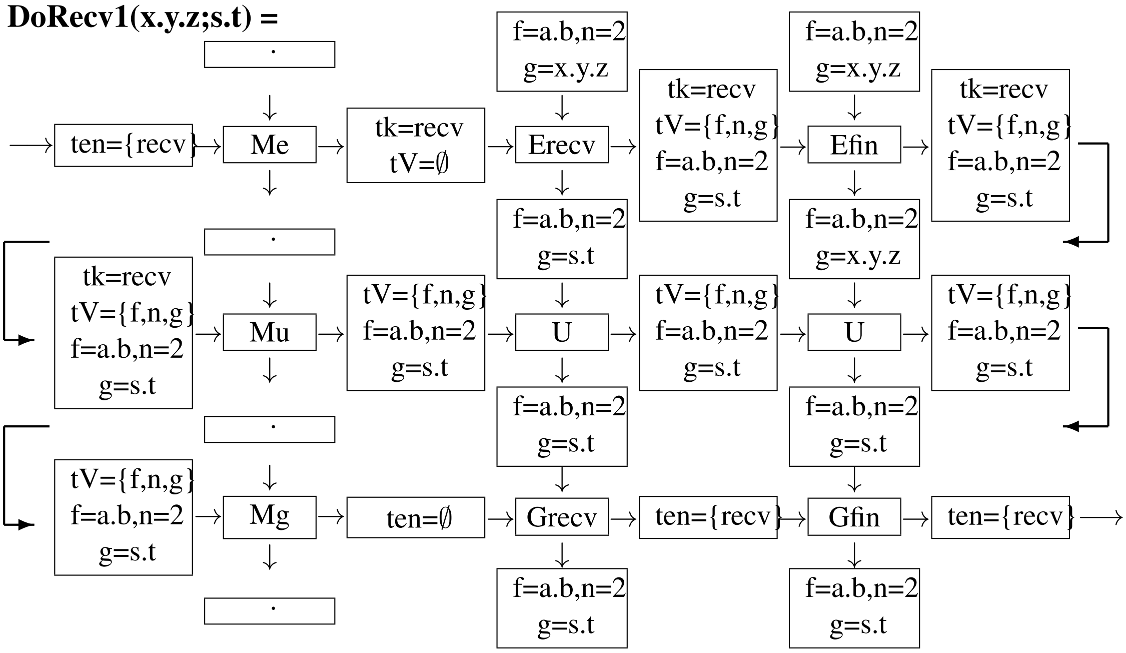
The first piece of the scenario is an initialization step `Init1(x.y.z)`. This scenario starts with the given data `f,n` and by the random assignment `g=x.y.z` generates the initial data for all processes associated to the events, i.e., for the `recv` and `fin` processes. In addition, the scenario starts the checking of the guards' validity; in this case the guard of the `recv` event is true and `recv` is exported on the last line.

Init1(x.y.z) =



The next piece of the running scenario `DoRecv1(x.y.z;s.t)` corresponds to the application of the `recv` event resulting in a state change with `g=x.y.z` replaced by `g=s.t`. It is coordinated by the manager process which coordinates in turn the `Me` (applying event actions), `Mu` (updating the cash copies of all data in the processes with the new values) and `Mg` (checking the guards again) activities.

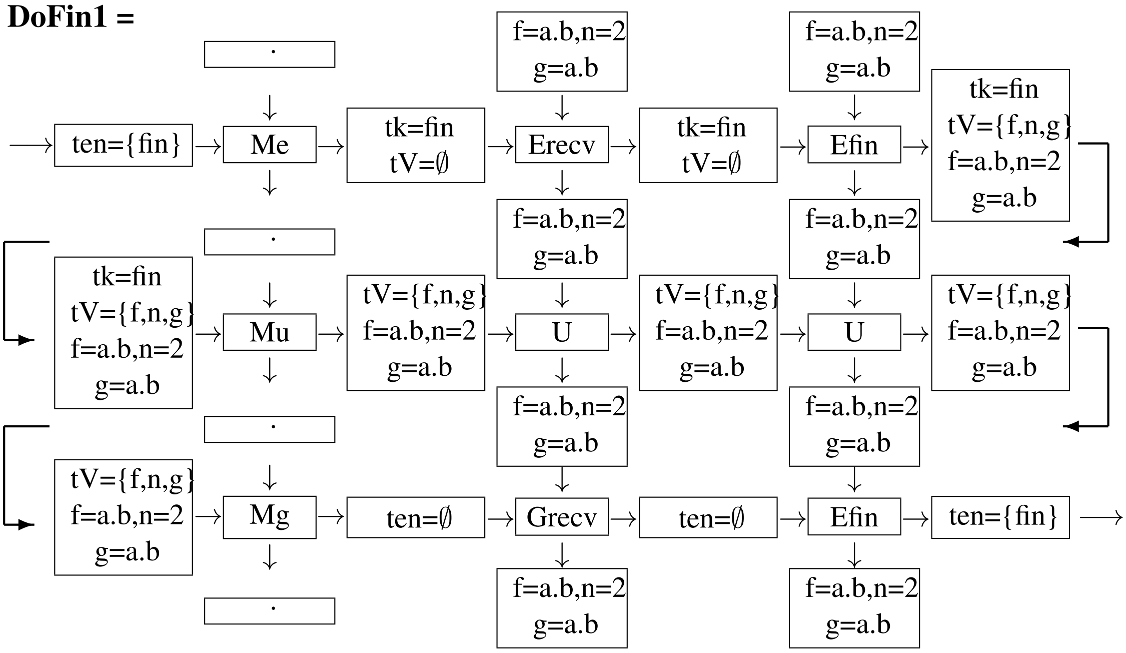
DoRecv1(x.y.z;s.t) =



Hopefully, after a number of such steps the random assignment gets `g=a.b` and in that case the exported true guard in the last line is `fin`, not `recv`.

If the `fin` guard is true, then the following last piece of scenarios applies. In this part, the `fin` event has no actions, so nothing changes in the states. Therefore this piece of the scenario is repeated forever.

DoFin1 =



Protocol first refinement

Refined Event-B model: FTP, 1st refinement In the first refinement, we modify the event `receive` in order to send to the receiver concrete parts of file f. Event `receive` will no longer produce randomly files until it obtains one equal with f. Instead, it will send one element of the file f at each step. For this we need to introduce a new variable r which represents an index of the file f. At each step, the r-th element of f is copied in the file g of the receiver's site. The file transfer is finished when r is greater than n.

The first refinement of our model in Event-B has the following events:

FTP-EB2 ::=

```
init
  $g := \emptyset$
  $r := 1$
```

```
receive
 when
  $r \leq n$
 then
  $g := g \cup \{r \mapsto f(r)\}$
  $r := r + 1$
 end
```

```
final
 when
  $r = n + 1$
 then
  $skip$
 end
```

Translation of the Event-B model: FTP, 1st refinement The corresponding rv-IS specification FTP-IS2 use the same formula as in the case of the initial model, but with modules `I1, Grecv1, Gfin1, Erecv1` slightly changed, i.e., they are replaced by the new modules `I2, Grecv2, Gfin2, Erecv2` listed in Table 2.5.

```
module I2
{listen nil}{read f,n}{
  g =∅;
  r = 1;
  tV = {f,n,g,r};
}{speak tV}{write nil}
```

```
module Erecv2
{listen tk,tV}{read V}{
  if(tk = Erecv){
    g = g U {r ↦ f(r)};
    r = r+1; tV = V;};
}{speak tk,tV}{write V}
```

```
module Grecv2
{listen ten}{read V}{
  if(r ≤ n){
    ten = ten U {Erecv};};
}{speak ten}{write V}
```

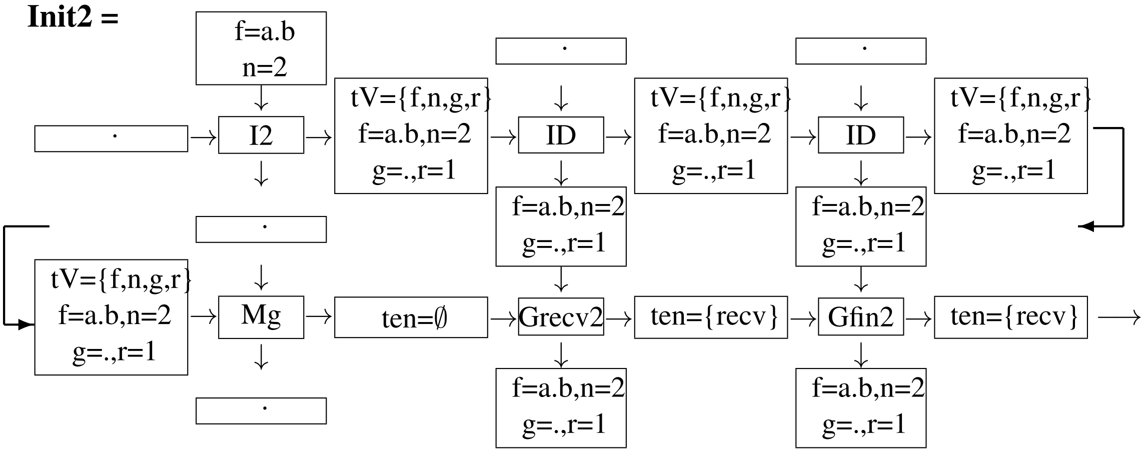
```
module Gfin2
{listen ten}{read V}{
  if(r = n+1){
    ten = ten U {Efin};};
}{speak ten}{write V}
```

Table 2.5: Specific modules for FTP-IS2 specification

Let us analyze the above case again: suppose that `f=a.b,n=2`. The running scenario is unique (deterministic) this time and consists of an initial action, followed by `n` times repeated `recv` actions, followed by repeated `fin` actions.

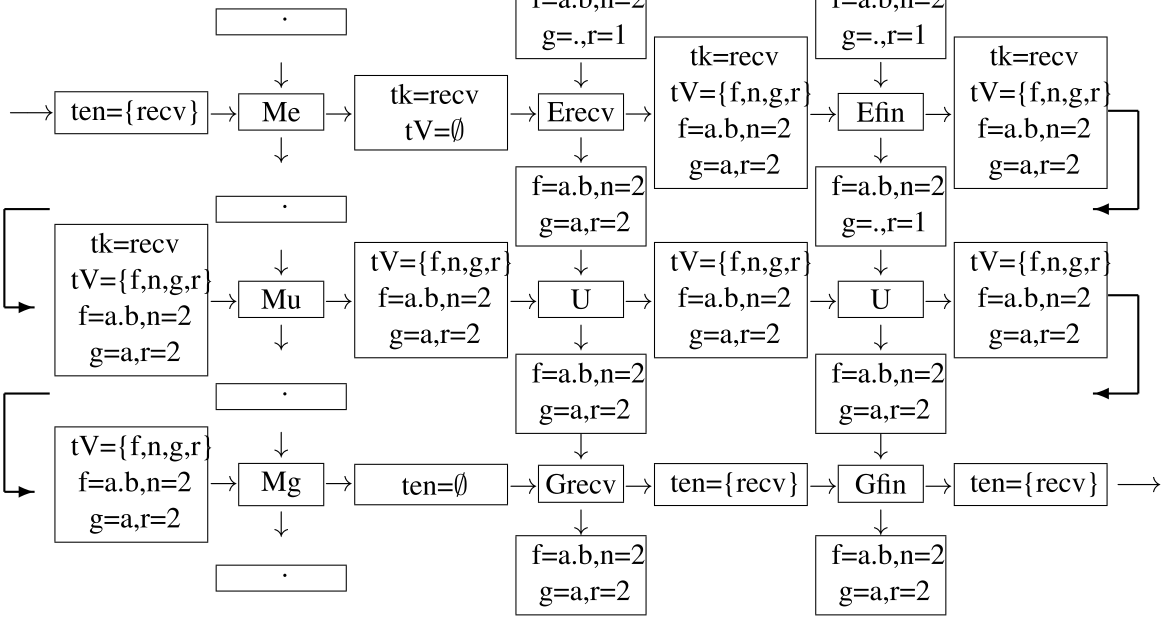
The first piece of the scenario is the initialization step `Init2`. It starts with the given data `f,n` and generates the initial data for all processes using the empty file for g, denoted g=.; moreover, r=1. Is also checks the guards and the true guard recv is exported on the last line.

Init2 =



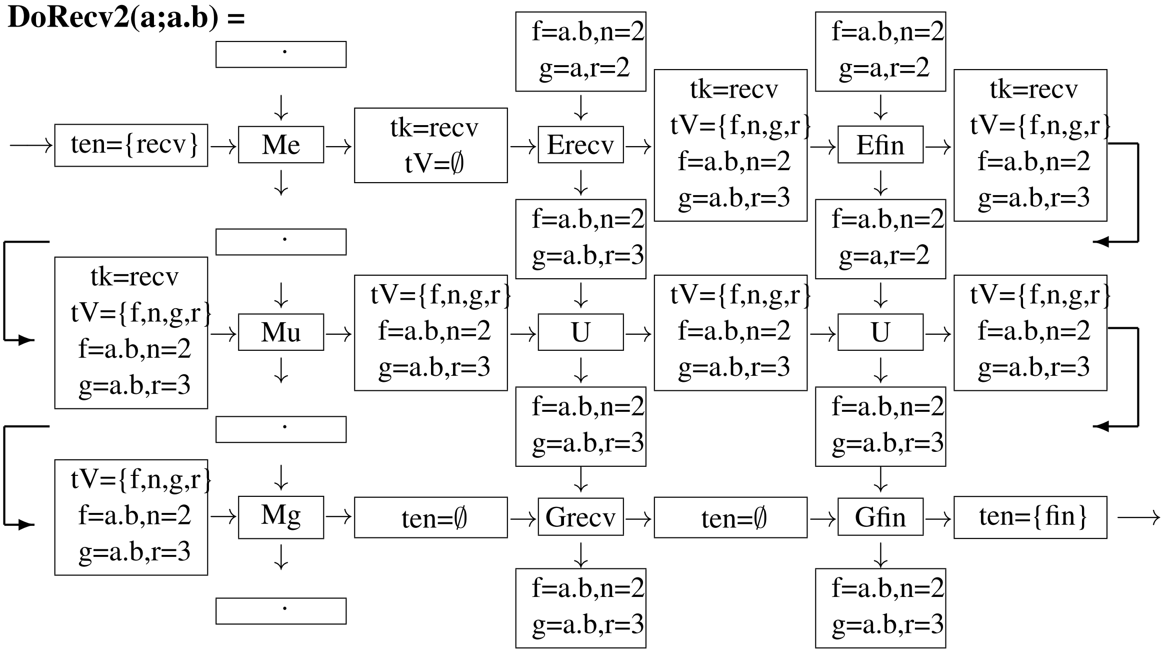
The next part of the scenario corresponds to the modeling of the recv actions. The modeling of the assignment g=a, corresponding to the increment of g=. with the item f(1)=a is described below.

DoRecv2(.;a) =



Next is the modeling of the assignment g=a.b, corresponding to the increment of g=a with the item f(2)=b.

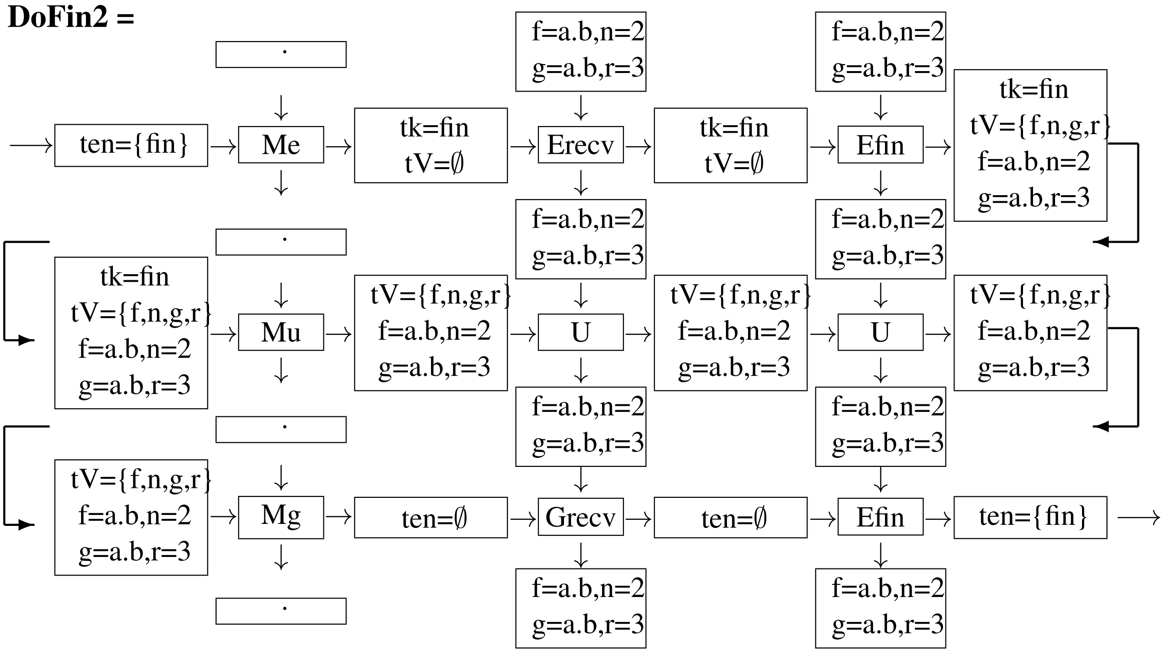
DoRecv2(a;a.b) =



·
f=a.b,n=2 g=a,r=2
f=a.b,n=2 g=a,r=2
ten={recv}
Me
tk=recv tV=∅
Erecv
tk=recv tV={f,n,g,r} f=a.b,n=2 g=a.b,r=3
Efin
tk=recv tV={f,n,g,r} f=a.b,n=2 g=a.b,r=3
·
f=a.b,n=2 g=a.b,r=3
f=a.b,n=2 g=a,r=2
tk=recv tV={f,n,g,r} f=a.b,n=2 g=a.b,r=3
Mu
tV={f,n,g,r} f=a.b,n=2 g=a.b,r=3
U
tV={f,n,g,r} f=a.b,n=2 g=a.b,r=3
U
tV={f,n,g,r} f=a.b,n=2 g=a.b,r=3
·
f=a.b,n=2 g=a.b,r=3
f=a.b,n=2 g=a.b,r=3
tV={f,n,g,r} f=a.b,n=2 g=a.b,r=3
Mg
ten=∅
Grecv
ten=∅
Gfin
ten={fin}
·
f=a.b,n=2 g=a.b,r=3
f=a.b,n=2 g=a.b,r=3

After these two macro-steps, the `fin` guard is true. Therefore, the last piece of the scenario corresponding to `fin` is applied and repeated forever.

DoFin2 =



Refinement preservation The state space of FTP-IS2 is $S2 = \{\mathtt{f}, \mathtt{n}, \mathtt{g}, \mathtt{r}\}$, while that of FTP-IS1 is $S1 = \{\mathtt{f}, \mathtt{n}, \mathtt{g}\}$. The class space of FTP-IS2 is $C2 =$

$\{\mathtt{ten}, \mathtt{tk}, \mathtt{tV} = (\mathtt{f}, \mathtt{n}, \mathtt{g}, \mathtt{r})\}$, departing from that of FTP-IS1 $C1 = \{\mathtt{ten}, \mathtt{tk}, \mathtt{tV} = (\mathtt{f}, \mathtt{n}, \mathtt{g})\}$ by the type of `tV`.

Fact 2.1.1 *FTP-IS2 is a refinement of FTP-IS1.*

Proof: (Sketch) Let $\rho = (\rho_s, \rho_c)$ be a relation between the states and classes of FTP-IS2 and FTP-IS1, where $\rho_s : S2 \to S1$ and $\rho_c : C2 \to C1$ are the natural projections which makes abstraction of `r`. If `Scen` is a scenario in FTP-IS2, then $\rho(\mathtt{Scen})$ is a scenario in FTP-IS1.

Moreover, the weak relative deadlock freedom is valid: if `Scen1` is a partial scenario in FTP-IS2 and the scenario $\rho(\mathtt{Scen1})$ can be extended in FTP-IS1, then the same is true for `Scen1` in FTP-IS2. □

Protocol second refinement

Refined Event-B model: FTP, 2nd refinement Since our goal is a distributed execution of this file transfer protocol, the first refinement does not satisfy our needs. In the first refinement, the receiver has access to the file *f* which is supposed to be on the sender's site.

In the second refinement, we will avoid this disadvantage. We introduce a new event `send` that we can think, informally, to be triggered by the sender. Event `send` will appear before each occurrence of the event `receive`. In event `send`, the sender will send an item from file *f* which will be received in event `receive` by the `receiver`.

In order to assume a correct file transfer, we introduce a new variable s, a local counter on the sender's site, which represents the index of the next item to be sent to the receiver.

When it is safe to transfer a data from the sender to the receiver, the data item d (also a new variable), which is equal to $f(s)$, and the counter s incremented are send to the receiver by event `send`. Notice that the sender waits for an acknowledgement from the receiver in order to assure that the transfer is safe. This acknowledgement is counter r.

Event `receive` checks if the received counter s is different from its counter r and, if so, it accepts the item d, that will be stored in its file g, and increments the counter r.

The Event-B events that encode the informal behavior of the protocol as described above are:

FTP-EB3 ::=

```
init
  $g = \emptyset$
  $s = 1$
  $r = 1$
  $d :\in D$
```

```
send
  when
    $s = r$
    $r \neq n + 1$
  then
    $d = f(s)$
    $s = s + 1$
  end
```

```
receive
  when
    $s = r + 1$
  then
    $g = g \cup \{r \mapsto d\}$
    $r = r + 1$
  end
```

```
final
  when
    $r = n + 1$
  then
    skip
  end
```

```
module I3
{listen nil}{read f,n}{
  g =∅; r = 1; s = 1;
  d :∈ D; tV = {f,n,g,r,d};
}{speak tV}{write nil}
```

```
module Grecv3
{listen ten}{read V}{
  if(s = r+1){
    ten = ten U {Erecv};};
}{speak ten}{write V}
```

```
module Gsend3
{listen ten}{read V}{
  if(s = r & r ≠ n+1){
    ten = ten U {Esend};};
}{speak ten}{write V}
```

```
module Erecv3
{listen tk,tV}{read V}{
  if(tk = Erecv){
    g = g ∪ {r ↦ d}; r = r+1;
    tV = V;};
}{speak tk,tV}{write V}
```

```
module Esend3
{listen tk,tV}{read V}{
  if(tk = Esend){
    d = f(s); s = s+1;
    tV = V;};
}{speak tk,tV}{write V}
```

Table 2.6: Specific modules for FTP-IS3 specification

Translation of the Event-B model: FTP, 2nd refinement For presenting the associated rv-IS specification FTP-IS3, first we fix the following set of events $\mathrm{Ev} = \{\mathtt{recv}, \mathtt{send}, \mathtt{fin}\}$. The specification is given by the following expression:

```
FTP-IS3 ::=    (I3 # ID # ID # ID)
             $ (Mg # Grecv3 # Gsend3 # Gfin2)
             $ while_st(ten ≠ ∅) {
                     (Me # Erecv3 # Esend3 # Efin)
                   $ (Mu # U # U # U)
                   $ (Mg # Grecv3 # Gsend3 # Gfin2)
               }
```

As in the case of the first refinement, we list in Table 2.6 only the new modules.

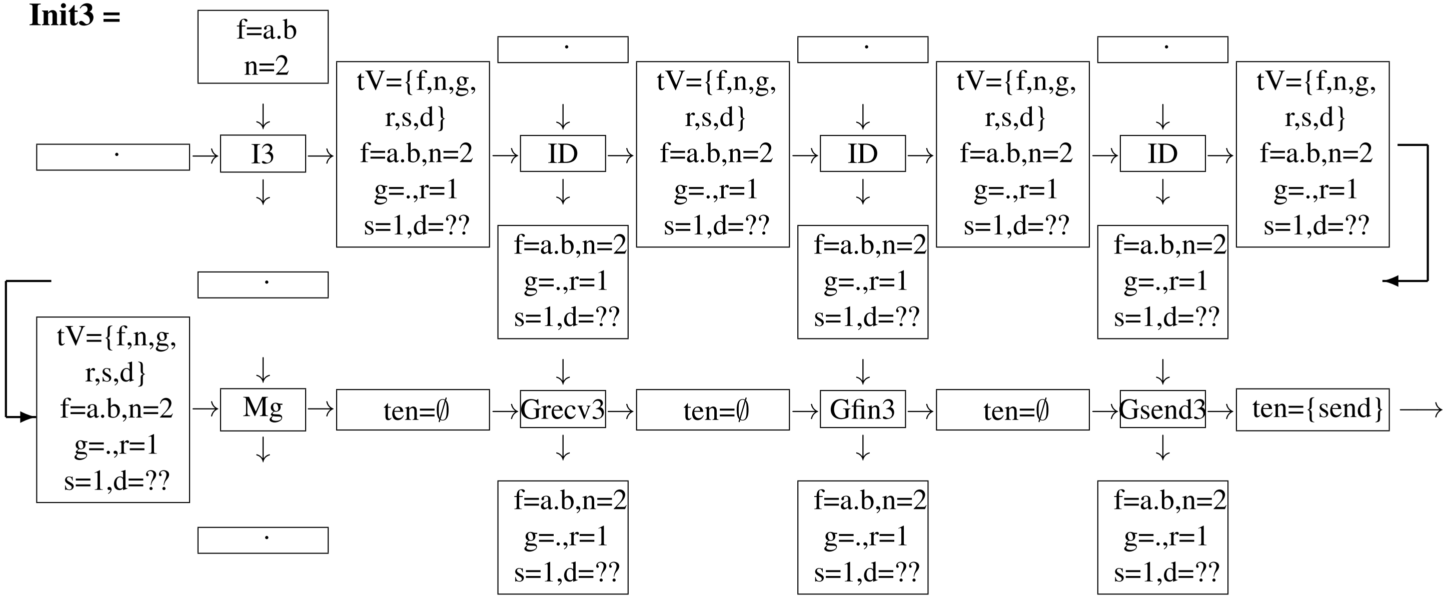
The scenarios can be constructed in a similar way as in previous subsubsections. One particular example, for the particular file `f=a.b` we have used so far, is described below.

The scenario pieces Init3, DoSend3(a) and DoRecv3(a) we will be talking about in the next comments are presented in full details in the next three pages.

The scenario starts with the initialization step Init3. The input state has the given data f,n. The scenario describes the setting of the initial data for all processes using the empty file for g and a random datum d=?? for the send channel. Therefore, at the beginning all processes have the state f=a.b, n=2, g=., r=1, d=??, s=1. The manager also checks the guards and the single true guard send is exported on the last line.

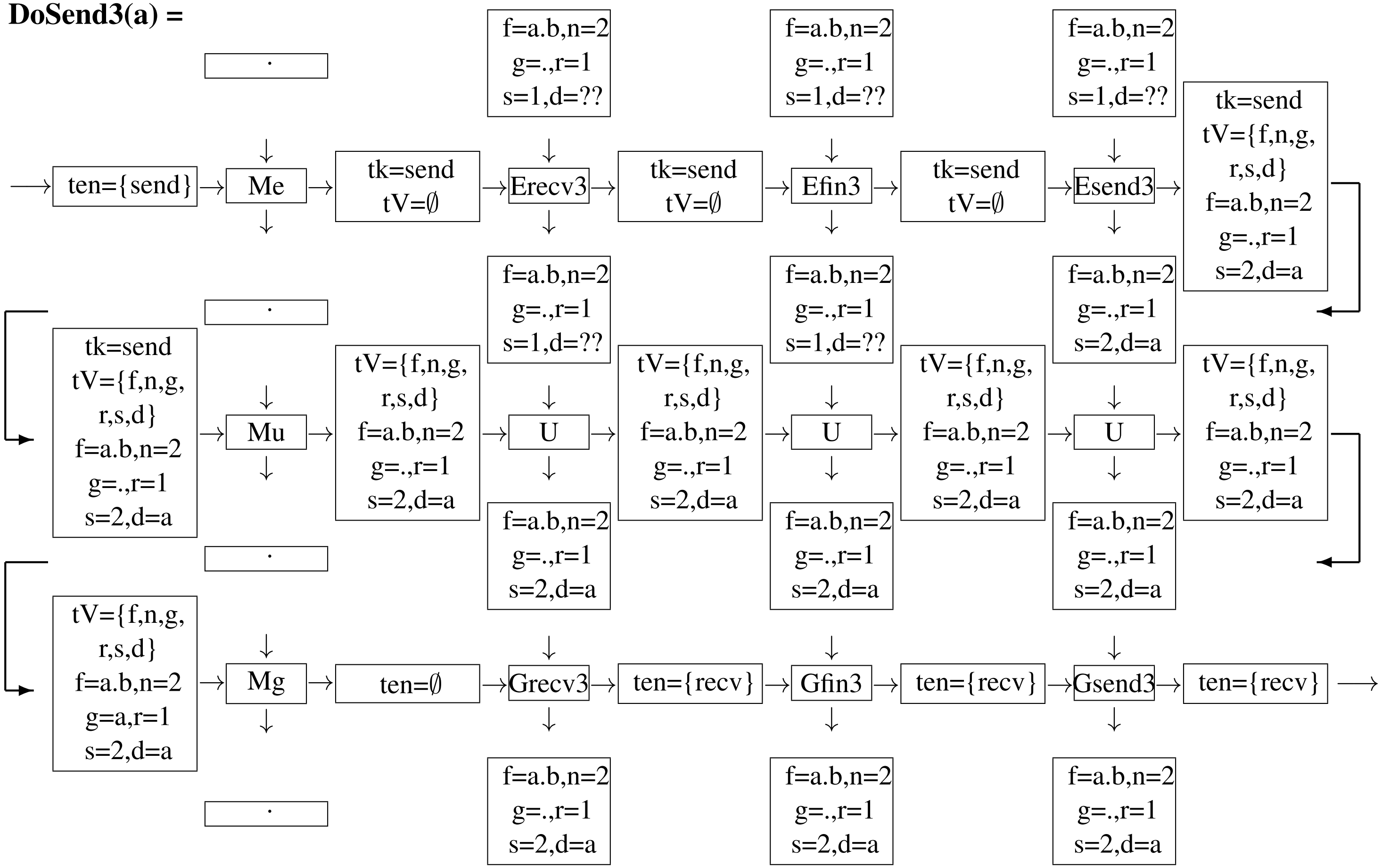
The next part of the scenario corresponds to the modeling of an alternate application of send and recv actions. It begins with DoSend3(a) describing the modeling of the send assignment corresponding to the load of f(s) (equal to a) in d and the increment of s. The next part of the scenario, denoted DoRecv3(a), corresponds to the modeling of the recv actions. This part contains the modeling of the assignment g=a, corresponding to the increment of g=. with the value of d.

Init3 =



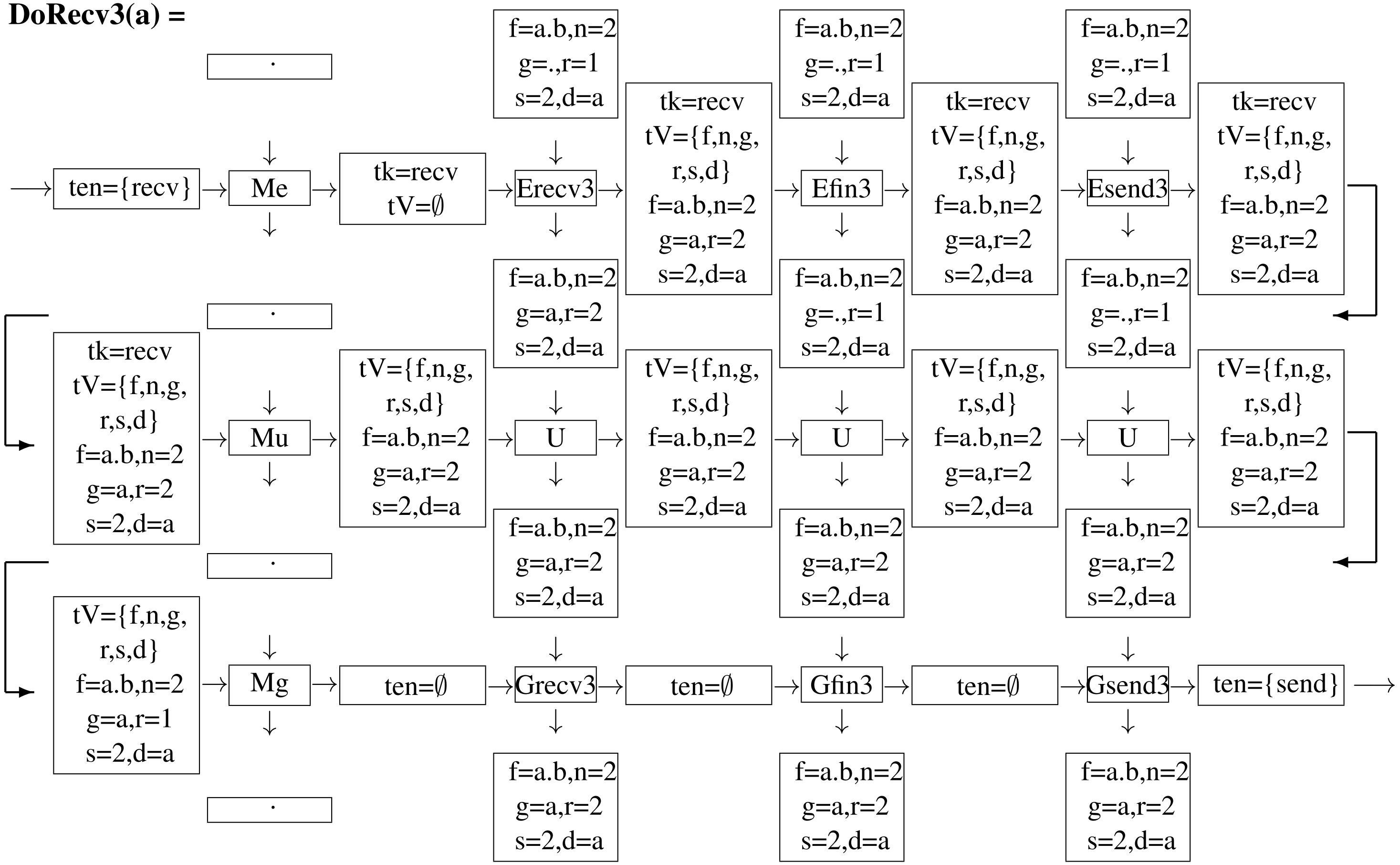
f=a.b
n=2
·
I3
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=.,r=1
s=1,d=??
·
ID
f=a.b,n=2
g=.,r=1
s=1,d=??
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=.,r=1
s=1,d=??
·
ID
f=a.b,n=2
g=.,r=1
s=1,d=??
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=.,r=1
s=1,d=??
·
ID
f=a.b,n=2
g=.,r=1
s=1,d=??
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=.,r=1
s=1,d=??
·
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=.,r=1
s=1,d=??
Mg
·
ten=∅
Grecv3
f=a.b,n=2
g=.,r=1
s=1,d=??
ten=∅
Gfin3
f=a.b,n=2
g=.,r=1
s=1,d=??
ten=∅
Gsend3
f=a.b,n=2
g=.,r=1
s=1,d=??
ten={send}

DoSend3(a) =



ten={send}
Me
tk=send tV=∅
Erecv3
tk=send tV=∅
Efin3
tk=send tV=∅
Esend3
tk=send tV={f,n,g, r,s,d} f=a.b,n=2 g=.,r=1 s=2,d=a
f=a.b,n=2 g=.,r=1 s=1,d=??
f=a.b,n=2 g=.,r=1 s=1,d=??
f=a.b,n=2 g=.,r=1 s=1,d=??
f=a.b,n=2 g=.,r=1 s=1,d=??
f=a.b,n=2 g=.,r=1 s=1,d=??
f=a.b,n=2 g=.,r=1 s=2,d=a
tk=send tV={f,n,g, r,s,d} f=a.b,n=2 g=.,r=1 s=2,d=a
Mu
tV={f,n,g, r,s,d} f=a.b,n=2 g=.,r=1 s=2,d=a
U
tV={f,n,g, r,s,d} f=a.b,n=2 g=.,r=1 s=2,d=a
U
tV={f,n,g, r,s,d} f=a.b,n=2 g=.,r=1 s=2,d=a
U
tV={f,n,g, r,s,d} f=a.b,n=2 g=.,r=1 s=2,d=a
f=a.b,n=2 g=.,r=1 s=2,d=a
f=a.b,n=2 g=.,r=1 s=2,d=a
f=a.b,n=2 g=.,r=1 s=2,d=a
tV={f,n,g, r,s,d} f=a.b,n=2 g=a,r=1 s=2,d=a
Mg
ten=∅
Grecv3
ten={recv}
Gfin3
ten={recv}
Gsend3
ten={recv}
f=a.b,n=2 g=.,r=1 s=2,d=a
f=a.b,n=2 g=.,r=1 s=2,d=a
f=a.b,n=2 g=.,r=1 s=2,d=a

DoRecv3(a) =



·
ten={recv}
Me
tk=recv
tV=∅
·
·
·
tk=recv
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=a,r=2
s=2,d=a
Mu
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=a,r=2
s=2,d=a
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=a,r=1
s=2,d=a
Mg
ten=∅
f=a.b,n=2
g=.,r=1
s=2,d=a
Erecv3
f=a.b,n=2
g=a,r=2
s=2,d=a
U
f=a.b,n=2
g=a,r=2
s=2,d=a
Grecv3
f=a.b,n=2
g=a,r=2
s=2,d=a
tk=recv
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=a,r=2
s=2,d=a
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=a,r=2
s=2,d=a
ten=∅
f=a.b,n=2
g=.,r=1
s=2,d=a
Efin3
f=a.b,n=2
g=.,r=1
s=2,d=a
U
f=a.b,n=2
g=a,r=2
s=2,d=a
Gfin3
f=a.b,n=2
g=a,r=2
s=2,d=a
tk=recv
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=a,r=2
s=2,d=a
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=a,r=2
s=2,d=a
ten=∅
f=a.b,n=2
g=.,r=1
s=2,d=a
Esend3
f=a.b,n=2
g=.,r=1
s=2,d=a
U
f=a.b,n=2
g=a,r=2
s=2,d=a
Gsend3
f=a.b,n=2
g=a,r=2
s=2,d=a
tk=recv
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=a,r=2
s=2,d=a
tV={f,n,g,
r,s,d}
f=a.b,n=2
g=a,r=2
s=2,d=a
ten={send}

There are two more macro steps modeling of the assignment `g=a.b`, corresponding to the load of the value `f(s)=b` in `d` and the increment of `g=a` with the new value of `d`, namely `b`.

After these two steps, the `fin` guard is true. Therefore, the last piece of scenario corresponding to `fin` is applied and repeated forever.

The partial scenarios for sending/receiving 'b' **DoSend3(b), DoRecv3(b)** and for **DoFin2** can be computed in a similar way.

Refinement preservation The state space of FTP-IS3 is $S3 = \{\mathtt{f}, \mathtt{n}, \mathtt{g}, \mathtt{r}, \mathtt{s}, \mathtt{d}\}$, while that of FTP-IS2 is $S1 = \{\mathtt{f}, \mathtt{n}, \mathtt{g}, \mathtt{r}\}$. The class space of FTP-IS2 is $C3 = \{\mathtt{ten}, \mathtt{tk}, \mathtt{tV} = (\mathtt{f}, \mathtt{n}, \mathtt{g}, \mathtt{r}, \mathtt{s}, \mathtt{d})\}$ and that of FTP-IS1 is $C2 = \{\mathtt{ten}, \mathtt{tk}, \mathtt{tV} = (\mathtt{f}, \mathtt{n}, \mathtt{g}, \mathtt{r})\}$. FTP-IS3 has a new event `send` and the sets used for $\mathtt{ten}, \mathtt{tk}$ are larger, including this new element.

Fact 2.1.2 *FTP-IS3 is a refinement of FTP-IS2.*

Proof: (Sketch) The Let $\rho = (\rho_s, \rho_c)$ be a relation between the FTP-IS3 and FTP-IS2, where $\rho_s : S2 \to S1$ and $\rho_c : C2 \to C1$ are the natural projections which makes abstraction of $\mathtt{s}, \mathtt{d}, \mathtt{send}$. If `Scen` is a scenario in FTP-IS3, then $\rho(\mathtt{Scen})$ is a scenario in FTP-IS2 up to sub-scenario stuttering corresponding to the application of the macro-steps associated to the *send* event and of the column corresponding to the *send* event. Indeed, this latter sub-scenarios have no visible effect on the states and classes of FTP-IS2 from their border.

Moreover, the weak relative deadlock freedom is valid: if `Scen1` is a partial scenario in FTP-IS2 and the scenario $\rho(\mathtt{Scen1})$ can be extended in FTP-IS2, then the same is true for `Scen1` in FTP-IS3. □

2.1.8 Final comments

The main contributions of this section are:

1. the introduction of a scenario-based definition of refinement of register-voice interactive systems (rv-IS models); and
2. the definition of a translation EB2IS from Event-B models to structured rv-IS models.

In addition,

3. we have provided enough evidence that the translation preserves refinement by considering a chain of complex enough refined Event-B models and the translated chain of rv-IS models.

From our future research plans, directly related to the research presented in this section, we emphasize the following:

1. provide a general mathematical proof that the translation EB2IS preserves refinement;

2. give a more concise definition of rv-IS refinement based on the rv-IS models themselves, not on the associated scenarios;

3. find another mathematical proof that the EB2IS translation preserves refinement, now using the refinement definition based on models, not on traces and scenarios.

2.2 Work on a real-time extension for Event-B

This work on real-time heavily relies on the proposed integration of Event-B and rv-IS formalisms. We investigated the two following research topics:

– The introduction and development of temporal pointers in rv-IS. A temporal pointer specifies a time address on a stream. Their introduction and use is based on usual pointers and the space-time duality transformations. In particular, rv-IS interaction composition operators are used to deal with real-time constraints.

– The second objective is to develop rv-IS inspired real-time Event-B models. In order to see the adequacy of the enrichment of Event-B with this time formalism we have suited real-time aspects of a NoC (Network-on-Chip) Event-B modelling.

2.2.1 Introduction

Modelling time-dependent computing in Event-B - a state-based formal method based on refinement - is currently of industrial interest with immediate applications in industrial processing. Furthermore, time-awareness is displayed by a large plethora of contemporary systems such as GPS applications, sensor networks, network-on-chip (NoC) computing, etc. We explore time modelling in Event-B via the related structured interactive systems formalism, where both spatial and temporal reasoning are already available in the context of a flexible structuring of component interactions. In this paper we introduce pointers in structured interactive systems, in particular temporal pointers that allow for finer-grained synchronization between parallel processing. We discuss the impact of temporal pointers on Event-B modelling and illustrate our findings on a small case study for congestion-awareness in NoCs.

Timing aspects are important in industrial processing, for instance to express deadlines throughout the development process. Moreover, time-awareness is displayed by a large plethora of contemporary systems such as GPS applications, sensor networks, network-on-chip (NoC) computing, etc. If we are able to express timing aspects and time-awareness via precise modeling, such as provided by formal methods, then we are able to analyze our timed systems and consequently understand their strengths and weaknesses.

Formal methods, with their mathematic proving core, are an important instrument in modeling and analyzing software-intensive systems. Traditionally characterized as hard to use, due to the requested mathematical background and the lack of automatic tools, nowadays formal methods have matured, to the point where they are considered in industry when developing software-intensive systems [64].

Examples of the industrial undertaking of formal methods are increasing. The famous line 14 of the driverless Parisian metro [32], developed in 1998 using the B-method [1], is the first notable example of a formal method-based development, reviewed in [19]. The method used by Siemens for developing the software controlling the line 14 train ensured its correctness in a mathematical manner that effectively eliminated the unit testing from the software lifecycle. No human resources are now needed to operate the trains and in addition, the trains are faster, hence fewer are needed in total.

More recent examples of the Event-B [3] formal method usage in industry can be seen for instance with Space Systems [130] and SAP [11]. In Event-B, the development of a model is carried out step by step from an abstract specification to more concrete implementations. Using the *refinement* approach, a system can be described at different levels of abstraction, and the consistency in and between levels can be proved mathematically. Event-B has an associated tool, the Rodin Platform [59, 4] where proof obligations are generated automatically and proven either automatically or interactively. Thus, one gets a mathematical proof that the model is correct; this kind of integrated tool support is a major advantage of Event-B. Even though the envisioned industrial tool support assumes that the (difficult) mathematical modeling is hidden in the background, Rodin is certainly a big step forward in promoting formal methods to industry.

There are already several approaches to model time and time properties in Event-B. Patterns that permit the analysis of timing properties relating to the resilience and consistency of (SAP) business processes are presented in [10], based on an existing Event-B pattern for modelling time [12]. In [20], a formalism for the modelling and analysis of dynamic reconfiguration of dependable real-time systems is proposed. The timing properties of deadline, delay, and expiry are investigated in [46], by proposing several refinement patterns on using Event-B constructs for these properties consistently throughout the development. In [15], a process-based view from an Event-B model is extracted and augmented with timing constraints to form a timed automata model, to be an input for the Uppaal tool [8]. Another more recent approach that compares the execution in Event-B and Uppaal, in order to propose a notion of refinement for timed systems in Uppaal, is presented in [9]. Moreover, embedding time in Event-B based on an earlier modeling of discrete control over continuous evolution in the action systems [7] formalism is introduced in [5].

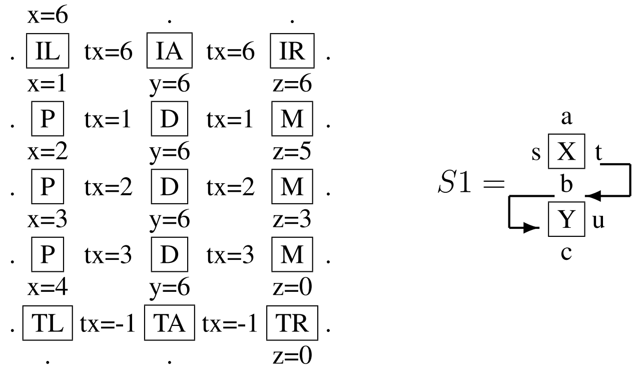
While the latter approach above takes advantage of the semantical compatibility between Event-B and the Action Systems formalism [6, 63], in this paper we approach time modeling in Event-B via another related formalism, namely the *register-voice interactive systems* (rv-IS) [56, 45, 33, 34]. This is a recent proposal for developing software systems using both structural state-based as well as interaction-based composition operators. Both spatial and temporal reasoning are already available in the context of a flexible structuring of component interactions. The contribution of this paper consists in introducing pointers in rv-IS, in particular temporal pointers that allow for finer-grained synchronization between parallel processing. We explore the relevance of temporal pointers on Event-B modelling, considering a recently proposed translation between Event-B and rv-IS [13]. Moreover, we base this discussion on a small case study for congestion-awareness in NoCs, inspired by earlier NoC modeling in Event-B [17].

We proceed as follows. In Subsection 2.2.2 we present the fundamental features of rv-IS, to be able to introduce spatial and temporal pointers in Subsection 2.2.3. In Subsection 2.2.4 we model an approach to congestion-aware NoC computing with temporal pointers in rv-IS and in Subsection 2.2.5 we discuss the impact of this kind of time-awareness on Event-B. We conclude in Subsection 2.2.6.

2.2.2 Structured interactive systems

In this subsection we describe the structured interactive systems to the extent needed in this paper.

The rv-IS formalism is built on top of register machines, closing them with respect to a space-time duality transformation. Specifically, we use the model, the core programming language, the specification formalism and the analysis techniques developed for modeling, programming and reasoning about interactive computing systems by the last author and coworkers in the recent years, see [56, 45, 33, 34]. In the following, we shortly overview the approach.



(a) Scenario for **Perfect1** (b) “Back-arrows” Scenario

Figure 2.12: Scenarios

Scenarios. A *scenario* is a two-dimensional rectangular area filled in with identifiers and enriched with data around each identifier. In our interpretation, the columns correspond to processes, the top-to-bottom order describing their progress in time. The left-to-right order corresponds to process interaction in a non-blocking message passing discipline. This means that a process sends a message to the right, then it resumes its execution. *(Memory) states* are placed at the north and at the south borders of the identifiers and *(interaction) classes* are placed at the west and at the east borders of the identifiers. In Fig. 2.12 (a) we illustrate a scenario for deciding whether the number 6 is a perfect number (i.e., it is equal to the sum of its proper divisors). The modules in this figure are described shortly.

Spatio-temporal specifications. A spatio-temporal specification combines constraints on both spatial and temporal data. For the spatial data, we use the common data structures and their natural representations in memory. For representing temporal data we use streams: a *stream* is a sequence of data ordered in time and is denoted as $a_0{}^\frown a_1{}^\frown\ldots$, where $a_0, a_1, \ldots$ are the data lying on the stream at time $0, 1, \ldots$, respectively.

A *voice* is defined as the time-dual of a register. Voices are simple temporal structures, represented on streams, that hold natural numbers. The value of a voice can be modified in a location and then propagated within the system. A voice can be "listened to" at various locations, at each location the piece of stream representing the voice displaying a particular value. Voices may be implemented on top of a stream in a similar way registers are implemented on top of a Turing tape, for instance specifying their starting address and their length. Most of usual data structures have natural temporal representations. Examples includes timed booleans, timed integers (denoted `tInt`), timed arrays, timed lists, etc.

The notation $\otimes$ is used for the product of memory states, while $^\frown$ for the product of interaction classes; $\mathbb{N}^{\otimes k}$ denotes $\mathbb{N}\otimes\ldots\otimes\mathbb{N}$ (k terms) and $\mathbb{N}^{\frown k}$ denotes $\mathbb{N}^\frown\ldots{}^\frown\mathbb{N}$ (k terms); the associated "star" operations are denoted as $(\_\_^{\otimes})^*$ and $(\_\_^{\frown})^*$.

A simple *spatio-temporal specification* $S : (m, p) \to (n, q)$ is a relation $S \subseteq (\mathbb{N}^{\frown m} \times \mathbb{N}^{\otimes p}) \times (\mathbb{N}^{\frown n} \times \mathbb{N}^{\otimes q})$, where m (resp. p) is the number of input voices (resp. registers) and n (resp. q) is the number of output voices (resp. registers). More general spatio-temporal specifications may be introduced using complex interface types, not only registers and voices.

Syntax of structured rv-programs. The *type* of a *structured rv-program* P, denoted by

$$P : (w(P), n(P)) \to (e(P), s(P)),$$

collects the types at the west, north, east and south borders of its scenarios. In general, these are relatively complex types built up from boolean and integer types - see the concrete types used in Agapia v0.1 [33].

We define the syntax of structured rv-programs below:

```
P ::= X | P # P | P % P | P $ P
  | if(C) then {P} else {P} | while_t(C) {P}
  | while_s(C) {P} | while_st(C) {P}
```

The starting blocks for the construction of structured rv-programs are called *modules*. The syntax of a module is:

```
module **module_name**
{listen temporal_vars}{read
spatial_vars}{
  code
}{speak temporal_vars'}{write
spatial_vars'}
```

The operations on structured rv-programs are briefly described below. More details and examples may be found in [56, 33, 34].

1. Composition: Due to their two dimensional structure, programs may be composed horizontally and vertically, as long as their types agree. They can also be composed diagonally by mixing the horizontal and vertical composition.

- For two programs $\mathtt{P_i} : (w_i, n_i) \to (e_i, s_i)$, $i = 1, 2$, the *horizontal composition* $\mathtt{P_1}\#\mathtt{P_2}$ is well defined only if $e_1 = w_2$; the type of the composite is $(w_1, n_1 \otimes n_2) \to (e_2, s_1 \otimes s_2)$.

- Similarly, the *vertical composition* $\mathtt{P_1}\%\mathtt{P_2}$ is defined only if $s_1 = n_2$; the type of the composite is $(w_1{}^\frown w_2, n_1) \to (e_1{}^\frown e_2, s_2)$.

- The *diagonal composition* $\mathtt{P_1}\$\mathtt{P_2}$ is a derived operation - it connects the east border of $\mathtt{P_1}$ to the west border of $\mathtt{P_2}$ and the south border of $\mathtt{P_1}$ to the north border of $\mathtt{P_2}$; it is defined only if $e_1 = w_2$ and $s_1 = n_2$; the type of the composite is $(w_1, n_1) \to (e_2, s_2)$.

2. If: For the "if" operation, given two programs with the same type $\mathtt{P}, \mathtt{Q} : (w, n) \to (e, s)$, a new program `if(C) then {P} else {Q}` $: (w, n) \to (e, s)$ is constructed, for a condition `C` involving both, the temporal variables in w and the spatial variables in n.

3. While: There are three while statements, each being the iteration of the corresponding composition operation.

- For a program $\mathtt{P} : (w, n) \to (e, s)$, the statement `while_t(C){P}` is defined if $n = s$ and C is a condition on the variables in $w \cup n$. The type of the result is $((w^\frown)^*, n) \to ((e^\frown)^*, n)$.

- The case of *spatial while* `while_s(C){P}` is similar.

```
module M
{listen a}{read
n}{
  code
}{speak a}{write
s}
```

```
module I
{listen a}{read
nil}{
  ti:tInt; ti=1;
}{speak
ti,a}{write nil}
```

```
module M1
{listen
ti,a}{read n}{
  code; ti++;
}{speak
ti,a}{write s}
```

```
module E
{listen ti,a}{read
nil}{
  null;
}{speak a}{write
nil}
```

Table 2.7: Modules used for defining the "for" statement

- If `P` : $(w, n) \to (e, s)$, the statement `while_st(C){P}` is defined if $w = e$ and $n = s$ and C is a condition on $w \cup n$. The type of the result is $(w, n) \to (e, s)$.

Derived statements. Many usual programming idioms may be naturally extended in this setting. For instance,

```
for_s(tInt ti=1;ti<k;ti++){M}
```

denotes

```
I # while_s(ti<k){M1} # E
```

where `I` sets `ti=1`, `M1` adds to `M` the `ti++` statement, and `E` discards `ti`. The modules are illustrated in Table 2.7.

Operational semantics of structured rv-programs. The operational semantics is given in terms of scenarios. Scenarios are built up with the following procedure:

1. Each cell of the associated grid has as label a module name.

2. An area around a cell may have additional information. For example, if a cell has the information $\mathtt{x} = 2$, that means that in that area `x` is updated to be 2.

3. The scenario is built from the current rv-program by reducing it to simple compositions of spatio-temporal specifications w.r.t. the syntax of the program, until we reach basic blocks, e.g. modules.

Example. We illustrate the operational semantics with the following structured rv-program **Perfect1** verifying if a number n is perfect:

```
module IL{listen nil}{read n}{
 tInt tn = n; Int x =
1;}{speak tn}{write x}
```

```
module IA{listen tn}{read nil}{
 Int y = tn;}{speak tn}{write y}
```

```
module IR{listen tn}{read nil}{
 Int z = tn;}{speak nil}{write z}
```

```
module P{listen nil}{read x}{
 tInt tx = x; x =
x+1;}{speak tx}{write x}
```

```
module D{listen tx}{read y}{
 if(y % tx !=0){tx =
0;};}{speak tx}{write y}
```

```
module M{listen tx}{read z}{
 z = z - tx;}{speak nil}{write z}
```

```
module TL{listen nil}{read x}{
 tx = -1;}{speak tx}{write nil}
```

```
module TA{listen tx}{read y}{
 null;}{speak tx}{write nil}
```

```
module TR{listen tx}{read z}{
 null;}{speak nil}{write z}
```

Table 2.8: The modules of the **Perfect1** program

```
   (IL # IA # IR)
 % while_t(x =< n/2){P # D # M}
 % (TL # TA # TR)
```

The modules are listed in Table 2.8.

In our rv-program we can imagine that we have three processes: one generates all the numbers in the set $\{n/2, \ldots, 1\}$ (module `P`), one checks if a number is a divisor of n (module `D`) and the last one updates a variable z (module `M`). Modules `IL`, `IA` and `IR` are used for initializations and `TL`, `TA` and `TR` for termination. At the end of the program, if the variable z is 0, then the number n is perfect.

In order to show how we can construct a scenario for the rv-program above let us consider a concrete example for $n = 6$. The scenario for $n = 6$ is presented in Fig. 2.12 (a).

In the first line of the scenario we initialize the processes with the needed information: module `IL` is reading the value $n = 6$ and provides the first process with $x = 1$ and declare a temporal variant of n, namely $tn = 6$, that will be used by modules `IA` and `IR` for the other initializations; modules `IA` and `IR` use the temporal variable tn for initializing the other two processes with the initial value of n, namely $y = 6$, $z = 6$, respectively.

In the next step, module `P` produces a temporal data $tx = 1$ (tx is equal with the data x of the first process) and increases x. Module `D` verifies if tx is a divisor of y and, if it is not, then it resets the value of tx to 0. Finally, module `M` decreases the value of z by tx. Notice that module `M` decreases the value of z only with the divisors of the initial variable n. We continue this steps until the variable x becomes greater than $\frac{n}{2}$.

A final line contains terminating modules that rearrange some interfaces, keeping only the relevant result z.

2.2.3 Pointers in structured interactive systems

In this subsection we shortly introduce pointers in structured interactive systems. For brevity, the syntax and the semantics used for spatial and temporal pointer variables is not completely defined here. More details are given in our technical report [14].

Spatial and temporal pointers. The variables used for rv-IS spatial data types are extended to allow the use of pointers. These usual pointers are renamed here as *spatial pointers*. A spatial pointer variable specifies a physical address in the memory space. A (spatial) pointer is denoted by $^*x$; the value in the memory corresponding to the address specified by the pointer is accessed by the construct $\&x$. We use without formal introduction pointers and the basic pointer manipulation rules as they are used in common practice, for instance by C programmers.

Temporal pointers are similar constructs, but used for accessing temporal data. A *temporal pointer* represents a physical address on the interaction stream (a physical time) and is denoted by $^\wedge x$. To access the data value on the stream at the time address specified by the temporal pointer x we use the same notation $\&x$ as in the case of spatial pointers.

The enrichment of data types with pointers requires an explanation on the associated program semantics. In particular, it leads to a few restrictions on program operations, explained below.

Pointers in modules. The `module` construct may use both normal and pointer variables for their input and output interfaces. If a spatial pointer variable $^*s$ is present both in the input (read) and in the output (write) interfaces, then the corresponding pointer s refers to the same memory address before, during and after the execution of the module. Pointers that appear only in one, either output or input interface may be used to model allocation/deallocation of memory space.

Similarly, if a temporal pointer $^\wedge x$ is present in both the input (listen) and the output (speak) interfaces, then it refers to the same temporal address on the interaction stream before, during and after the execution of the module. In particular, the code in the module should be executed so that the computation and the spoken temporal output value $\& x$ at the output interface be finished in the same time cycle with the listening of the input data $\& x$ in the input interface. In other words, an occurrence of a pointer variable in both the input and the output interfaces of a module requires a synchronisation of the input listening and the output speaking events. Moreover, if there is a causal dependence between the output and the input values, then the computation for getting the output value has to be fast enough to produce the result in the same time cycle. Finally, temporal pointers that occur in one interface only can be used to handle starting or ending of scheduling synchronization constraints for stream processing.

In the presence of pointers, incorrect programs may easily occur. For instance, the code "`tInt ^x; &x=&x+&(x+1);`" can not be implemented: one has to output at the current time x a result that uses data that will be received later in time - at time $x + 1$ (a possible implementation may be based on a risky speculative computation: guess the output and verify later if it was a correct guess; if not, discard this computation scenario).

Before describing the use rules for pointer variables in structured interactive programming constructs, we emphasize a limitation in using pointer variables here: *pointer variables are allowed to be used locally for simple interface type (the types used for modules), but not for general interface type* - see Agapia v0.1 syntax [33] for a formal distinction between such types. For spatial pointers, this means we can not use pointers that refers to a memory space obtained by gluing together the memory space of two or more processes. Similarly, temporal pointers are not allowed to refer to a streaming interval that covers more than one transaction.

Pointers in structured interactive programming statements. Now, we consider the programming constructs. By the above observation, all programming constructs used for structured interactive programs, except for the diagonal (i.e., `$`) and the iterated diagonal (i.e., `while_st`) composition statements, are allowed to use also pointer variables at their interfaces. This actually means the following:

- The `if` statement can use pointer variables at all interfaces.
- The spatial program composition constructs # and `while_s` can use temporal pointer variables for the connecting (temporal) interfaces.
- The temporal program composition constructs % and `while_t` can use spatial pointer variables for the connecting (spatial) interfaces.

– The diagonal program composition constructs `$` and `while_st` can use either spatial or temporal pointer variables for the connecting interfaces, but not both types of pointers. Moreover, if spatial pointers are used, then the semantics of the diagonal composition operators has to be given with left “back-arrows”, as in Fig.2.12 (b), in order to preserve the original processes (i.e., to avoid process migration used by the default diagonal composition semantics). A similar observation applies if temporal pointers are used: in that case, we need to use a scenario semantics with vertical “back-arrows” for diagonal composition operators.

Actually, the only restriction in place is for the diagonal and the iterated diagonal compositions. Indeed, passing pointer variables via the connecting interfaces in a diagonal program composition requires to use spatial (resp. temporal) pointer variables over multiple processes (resp. transactions) and this is forbidden by the above convention.

For another, this time semantic reason consider a `while_st` statement with temporal pointer variables used for its iterated connecting temporal interface. Then, one may have a potentially unbounded loop of actions to be done at the same time cycle, which has to be avoided. (A similar constraint is used by many temporal computation models. For example, in the dataflow networks model the similar restriction says that no loop of actions is allowed if the actions are to be done in the same time cycle. The common solution is to insert delay actions within such loops.)

The Perfect program, revised. We illustrate the use of temporal pointers by considering a variation of the **Perfect1** program described in the previous subsection. In this new version, the divisibility check task is performed at various computing speeds due to the effect of the divisibility check implementation (for instance, if divisibility of n by x is implemented by a repeated subtraction of x from n `while(x>0){n=n-x};if(n=0){div=1}else{div=0}`, then the result is obtained faster for larger values of x) or due to an external reason (for instance, the runtime variation of the processor load).

In the implementation below we consider the first possibility mentioned above. The initial speed is set to be 1 and it is increasing by 1 with each transaction. In k steps, the program is processing in turn $1, 2, \ldots, k$ data, covering all the data in the range 1 to $\frac{n}{2}$ for $k = \lceil \frac{-1+\sqrt{1+4n}}{2} \rceil$. (Indeed, we observe that we have $\sum_{i=1}^{k} i = \frac{k(k+1)}{2} \approx \frac{n}{2}$.)

The new program **Perfect2** is

```
    (IL2 # IA # IR)
  % while_t(x =< n/2){P2 # D2 # M2}
  % (TL2 # TA # TR)
```

```
module IL2{listen nil}{read n}{
  tInt tn = n; Int x = 1; Int speed =
  1;
}{speak tn}{write x,speed}
```

```
module P2{listen nil}{read x,speed}{
  tInt tspeed = spead;
  tInt ^tx = t_alloc(speed *
  sizeof_tInt);
  for (i=0;i<speed;i++){&(tx+i)=x;
  x++;};speed++;
}{speak ^tx,tspeed}{write x,speed}
```

```
module D2{listen ^tx,tspeed}{read y}{
  for (i=0;i<tspeed;i++){
    if(y % &(tx+i) !=0){&(tx+i)=0;};
}{speak ^tx,tspeed}{write y}
```

```
module M2{listen ^tx,tspeed}{read z}{
  for (i=0;i<tspeed;i++){z = z -
  &(tx+i);}
}{speak nil}{write z}
```

```
module TL2{listen nil}{read x,speed}{
  tx = -1;
}{speak tx}{write nil}
```

Table 2.9: New modules of the **Perfect2** program

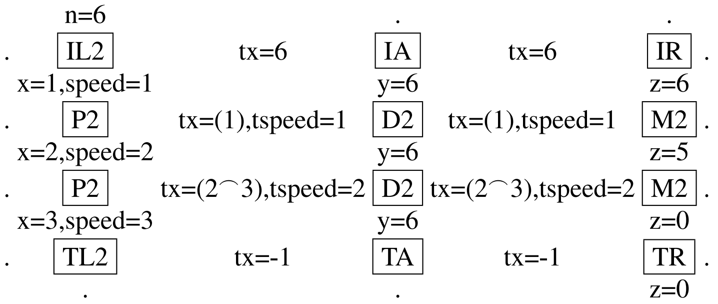
Its new modules are listed in Table 2.9 and a scenario in Fig. 2.13.

The scenario in Fig. 2.13 describes the case $n = 6$. The variable tx is a temporal pointer variable which can be used in the transaction using it. Progressively, there are streams with $1, 2, \ldots$ elements of type `tInt` allocated at this temporal pointer. For instance, in the 3rd row there is a temporal stream allocation of 2 `sizeof_tInt` elements; these elements are separated by '$\frown$'.

The variable tx is defined and it gets stream allocation in module $P2$. Later on, it is used by the modules $D2$ and $M2$. Being a temporal pointer, tx constraints all the modules using it to observe its temporal rules.

For instance, in the 3rd row of the scenario in Fig. 2.13 the following actions take place (ordered in time): (1) $P2$ speaks 2 at its output stream at the time specified by tx; then, $D2$ listens 2 and speaks 2 (the result of the divisibility check) at the same time specified by tx; and, finally, $M2$ listens this value of $D2$ at the same time tx and uses it the update the value of z. Notice that the semantics forces the implementation of the divisibility check be fast enough to return the result within the time cycle specified by tx. (2) Next, $P2$ speaks 3 at its output stream at the time corresponding to the pointer $tx + 1$; then, $D2$ listens 3 and speaks 3 (the result of the divisibility check) at the time corresponding to $tx + 1$;

and finally $M2$ listen this value at the same time and uses it to update z.



	n=6		.		.	
.	IL2	tx=6	IA	tx=6	IR	.
	x=1,speed=1		y=6		z=6	
.	P2	tx=(1),tspeed=1	D2	tx=(1),tspeed=1	M2	.
	x=2,speed=2		y=6		z=5	
.	P2	tx=($2^\frown 3$),tspeed=2	D2	tx=($2^\frown 3$),tspeed=2	M2	.
	x=3,speed=3		y=6		z=0	
.	TL2	tx=-1	TA	tx=-1	TR	.
	.		.		z=0	

Figure 2.13: A scenario for the **Perfect2** program

We emphasize the important difference in the execution semantics induces by the presence of pointers. Without the constraints induced by the temporal pointers, the common meaning of the execution is that a module such as P should be completely finished before starting the execution of a next module in the row, such as $D2$ here.

Eager evaluation semantics. To cope with the above informal semantics, one has to *add listen/read and speak/write statements into the modules code*. Then, an *eager evaluation semantics* can be develop for rv-programs: in this evaluation strategy, modules and other pieces of code needed for producing running scenarios start immediately and stop at the first listen/read statement waiting for the needed data values; when a data value is available, the run resumes and continues until a next stop where a needed value is not available; and so on.

However, there is a price to pay for this relaxed evaluation. An extra requirement for eager evaluation semantics is that on all possible runs all the variables in the output interfaces have to receive one and only one value via speak/write statements. This safeness strategy assures that all the variables in the output interfaces get appropriate values, while the variables in the input interfaces may or may not be used for producing them.

2.2.4 Case study: Congestion-aware NoC

In this subsection we present an rv-IS model of a congestion-aware NoC communication.

Informal presentation of the protocol. Consider a communication network with one-way communication channels, in both directions, between certain nodes.

For example, consider the network G with four nodes $\{1, 2, 3, 4\}$ and channels between the nodes $(1, 2)$, $(2, 3)$ and $(2, 4)$; the channels are denoted by $c12, c21, c23, c32, c24, c42$. The contents of a channel cij is considered to be a (bounded) queue of messages waiting to be processed by node j. The typical activity in a node i is the following:
| (i_{recv}) node i receives a message sent to it from one of its incoming channels;
| (i_{send}) node i inserts a new message it wants to sent via the network in an appropriate routing channel; and
| (i_{route}) node i routes a message from one of its incoming channels to an appropriate outgoing channel towards its destination.

Some part of network processing is presented in Fig. 2.10. Starting with a configuration $C0$ of the messages in transit lying in the communication channels, the processing does, in turn, one action for the nodes $1, 2, 3$ and 4 (in this order). We denote the messages sent from a node i to a node j as $a_{i,j}$ (the 1st), $b_{i,j}$ (the 2nd), and so on. In this example, in a first round node 1 inserts a message $a_{1,3}$, node 2 routes it, node 3 receives it, and node 4 inserts $a_{4,2}$. The routing is done along the shortest path, which in this example is unique for each pair of nodes. In a next round, node 1 receives $a_{2,1}$, node 2 receives $a_{3,2}$, nodes 3 and 4 select to receive messages, but there are no such messages in their incoming channels, so they do nothing.

Act	$c12$	$c21$	$c23$	$c32$	$c24$	$c42$	$Notation$
	$\emptyset$	$a_{2,1}{}^\frown a_{4,1}$	$\emptyset$	$a_{3,2}{}^\frown a_{3,4}$	$\emptyset$	$b_{4,1}$	$C0$
$1_{send} \rightarrow$	$a_{1,3}$	$a_{2,1}{}^\frown a_{4,1}$	$\emptyset$	$a_{3,2}{}^\frown a_{3,4}$	$\emptyset$	$b_{4,1}$	$C1$
$2_{route} \rightarrow$	$\emptyset$	$a_{2,1}{}^\frown a_{4,1}$	$a_{1,3}$	$a_{3,2}{}^\frown a_{3,4}$	$\emptyset$	$b_{4,1}$	$C2$
$3_{recv} \rightarrow$	$\emptyset$	$a_{2,1}{}^\frown a_{4,1}$	$\emptyset$	$a_{3,2}{}^\frown a_{3,4}$	$\emptyset$	$b_{4,1}$	$C3$
$4_{send} \rightarrow$	$\emptyset$	$a_{2,1}{}^\frown a_{4,1}$	$\emptyset$	$a_{3,2}{}^\frown a_{3,4}$	$\emptyset$	$b_{4,1}{}^\frown a_{4,2}$	$C4$
$1_{recv} \rightarrow$	$\emptyset$	$a_{4,1}$	$\emptyset$	$a_{3,2}{}^\frown a_{3,4}$	$\emptyset$	$b_{4,1}{}^\frown a_{4,2}$	$C5$
$2_{recv} \rightarrow$	$\emptyset$	$a_{4,1}$	$\emptyset$	$a_{3,4}$	$\emptyset$	$b_{4,1}{}^\frown a_{4,2}$	$C6$
$3_{recv} \rightarrow$	$\emptyset$	$a_{4,1}$	$\emptyset$	$a_{3,4}$	$\emptyset$	$b_{4,1}{}^\frown a_{4,2}$	$C7$
$4_{recv} \rightarrow$	$\emptyset$	$a_{4,1}$	$\emptyset$	$a_{3,4}$	$\emptyset$	$b_{4,1}{}^\frown a_{4,2}$	$C8$

Table 2.10: A typical step in the **Network** program

We point out a particularity of the network communication modelled this way. Due to nodes' linearisation, the messages $a_{i,j}$ with $i < j$ may have more receive and routing actions in a round than the messages $a_{i,j}$ with $i > j$. This produces a speedier processing of the network communications. If this is not the desired behaviour and a more symmetric behaviour is seeked, then one can (1) mark the messages which were already processed in the current round, (2) allow only unmarked messages to be further processed in the current round, and finally, (3) remove all marks at the end of the round.

An rv-IS model with temporal pointers. Now we describe a model **Network** with appropriate temporal pointers. The time information is local and reflects the streaming order of the messages in the channels cij.

The routing is static in this paper, hence no change of the routing protocol is made depending of the network congestion information. The network is also static: no nodes are allowed to join or leave the network. However, adaptations of our model to handle these dynamic aspects can be easily done. Finally, no state change is visible in this abstract model after the receiving or sending of messages. Nevertheless, the program below exports a minimal information on states `id,next` to which any additional state information can be added.

The program **Network** is

```
(I1 # for_s(ti=1;ti=<tn;ti++){I} # I2)
$ while_st(L!=empty){N1 # r=1
 # while_s(all (e_i,j - c_i,j) =< tC){
   r++
   # for_s(ti=1;ti=<tn;ti++){if(r%s_ti=0){N}}}
 # N2 }
```

Its modules are presented in Table 2.11. The variables used in this program have the following meaning. `G` denotes the network and `n` its number of nodes. `C` is the maximal number of precessing data in any communication channel before recomputing the nodes' processing speeds. `S` is the maximal slowdown. In other words: if the speed is 1, the node is active in any cycle; if the speed is `S` (the slowest speed), the node is active every `S` cycles. `r` counts the rounds. A node is active in a round `r` if this count is a multiple of the node's speed. `Next_node(tG,id,k)` computes the next routing node in `G` from `id` for a message with destination `k`. `New_speed` returns a value in `{1,...,S}`, representing the new processing rate to be used by a network's node.

Initially, the speed is at the slowest rate and all communication channels are empty.

Each channel `(i,j)` in `G` is a bounded queue. It is modelled as a temporal structure of messages `m_s,d`, sent from a source `s` to a destination `d`. We suppose each channel queue holds at most `C` items in a processing stage, before changing the processing speeding rates. A temporal pointer `c_i,j` points to this structure. There are two more temporal pointers associated to this channel: `b_i,j`, pointing to the first message in the queue and `e_i,j`, pointing to the first free position in the queue. The queue is empty if `b_i,j = e_i,j` and in that case `b_i,j` points to a fake element.

The passing of communication data from one stage to another is done via the temporal variable `L=(L_i,j)`. `L_i,j` is a stream [a temporal array may do this job too]. It collects the data lying in the channel `c_i,j` at the end of a processing stage; technically, these are the data in this communication channel between the pointers b_i,j and e_i,j.

A processing stage starts, in the module N1, by allocating C temporal positions for each channel c_i,j corresponding to an edge in G. Then, the data on the stream L_i,j are reallocated in order to be accessed by the pointers b_i,j (beginning of the message queue) and e_i,j (a free position after the ending of the message queue). Next, for each node k a new speed is computed, to be used in the current processing stage, by taking into account the quantity of the messages lying into the channels adjacent to k. In this implementation, it is assumed that a node can see how many messages are in its incoming and outgoing channels. At the end of a processing stage, an opposite activity is inserting, as described in module N2: the messages lying in a channel c_i,j between the pointers b_i,j and e_i,j are collected into the stream L_i,j.

The basic activity of a node is described in module N. Here, in each invocation of the module, the implementation randomly selects an activity: either it receives a message sent to it, or sends a new message, or routes a message from one of its incoming channels towards the destination. Such activities are performed if: (1) there are messages to be received or routed; and (2) there is space on the outgoing channels for the messages to be send or routed.

The adjustment of the processing rate to the nodes chosen speeds is implemented by the coordination activity specified by the Network program. If the speed of a node is set to be y, then the module N is activated for that node by the Network program once every y cycles.

The allocation policies for the temporal streaming data associated to the channels c_i,j is left unspecified, considering the lower level running environment to be in charge of it. It is not obvious from the code how different channels are mapped on a global virtual stream. In particular, the scheduling should respect the routing causality induced by the module N: if a message m_s,d is taken from the pointer b_in,id and routed at the pointer e_id,next(d), then this latter pointer should be scheduled after the former. This scheduling constraint may be pretty tough as the nodes processing rates may vary in time.

More deterministic implementations. We can also model a more specific scheduling information provided by the program. To this end, one can consider a global trace gt. Each action performed by an invocation of the module N will increase a pointer on this global trace by one. Instead of being an independent pointer structure, each c_i,j is to be mapped by the program in this global trace. In this case, the program itself may insure the routing causality. We discuss more on scheduling in our technical report [14].

2.2.5 The impact of temporal pointers on Event-B real-time modelling

Event-B [3] is a state-based formalism based on Action Systems [6, 63] and the B-Method [1]. In Event-B, the development of a model is carried out step by step from an abstract specification to more concrete implementations. Models in Event-B consist of *contexts* and *machines*. A context describes the static part of a model, containing sets and constants, together with axioms about these. A machine describes the dynamic part of a model, containing variables, invariants (boolean predicates on the variables), and events, that evaluate and modify the (shared) variables. Each event has an associated boolean predicate named *guard*, which determines if the event can execute (we say it is *enabled*) or not. Computation proceeds by repeated non-deterministic choice and execution of an enabled event. In [13], we have proposed a translation from Event-B to rv-IS. Essentially, each event is split into a guard module and an updating module; a manager module is responsible for determining what events are enabled; the shared memory reading and updating is simulated.

The NoC case studies in this paper are inspired by an approach to congestion-awareness in NoCs, modeled in Event-B in [17]. The work in [17] extends [16] and [18] where unicast and multicast NoC routing is addressed respectively, with Event-B. Congestion is treated in two ways in [17]. At an architectural level, when buffer congestion overcomes a certain threshold, the IP core voltage is increased, so that messages are treated faster and consequently removed from the buffers. Furthermore, the congestion-aware XYZ routing algorithm in a three dimensional NoC slightly modifies the deterministic XYZ algorithm: instead of messages being routed to their destination first on the X coordinate, then on the Y coordinate and finally on the Z coordinate, they are routed first on the coordinate where a non-congested buffer is detected.

In this paper, we model congestion-awareness in rv-IS with temporal pointers, based on a variant of changing the IP core speed (voltage). Our model here is intentionally more general and non-deterministic than in [17], in order to explore various modeling avenues. Analyzing the implications of the model presented in Subsection 2.2.4 on modeling time-awareness in Event-B, we reach several conjectures, outlined in the following.

First, we note that independent events in an Event-B model - events that read and update disjoint sets of variables - could execute in parallel (processes) if they could synchronize their outputs. This is clearly achievable with the temporal streams of rv-IS.

Second, the finer-grained synchronization we propose makes it possible to execute the same module several times at the same clock tick, when synchronizing the communication between modules with temporal pointers. This improves the efficiency of the model and can prove very beneficial to NoC computing.

Third, we could apply the same idea from our modeling in Subsection 2.2.4 to the Event-B modeling in [17]. Namely, we could model that, for energy saving, each IP core in a NoC adjusts its active/inactive status to the traffic it currently detects. Nodes are thus active only in certain cycles. This implies some global synchronization which is implicit anyway, such as for event guards evaluation to determine the next event(s) to execute.

2.2.6 Conclusions

In this paper we have proposed a finer-grained synchronization model for parallel processing in rv-IS, to be applied to Event-B modeling of timing aspects. Our approach is based on (temporal) pointers and is presented in more detail in our technical report [14]. Our concepts are illustrated with a small example on congestion-awareness NoC computing, a more simplistic version of which has been earlier presented in [17]. We believe that our approach has potential in providing an interesting approach to time modeling in Event-B, especially given our earlier introduced translation [13] between these two formalisms.

The rv-IS model is based on a space-time duality principle. Sometimes, this may look unconvincing, as, intuitively, one has the insight that time and space are quite different entities. The introduction of pointers appears to break down the space-time duality principle: for instance, with an extra space one can swap two memory elements in the same space, but it is not at all clear how to swap two stream values in the same temporal slots, using an extra time slot (such requirements may be specified with pointers, but not in the standard pointer-free rv-IS model). This simply shows that temporal pointers should be used carefully. In fact, even normal spatial pointers are considered harmful and avoided as much as possible by current programming practice. Hence, we think the same is true for temporal pointers.

```
module I1{listen nil}{read n,G,C,S}{tn=n; tG=G;
  tC=C; tS=S;}{speak tn,tG,tC,tS}{write nil}
```

```
module I{listen tn,tG,tC,tS,ti}{read nil}{
  id=ti; for(k=1;k=<tn;k++){
    if(k!=id){next(k)=Next_node(tG,id,k)}};
}{speak tn,tG,tC,tS,ti}{write id,next}
```

```
module I2{listen tn,tG,tC,tS}{read nil}{
  for all (i,j) edge in tG{ L_i,j = empty;}
}{speak tn,tG,L=(L_i,j),tC,tS}{write nil}
```

```
module N1{listen tn,tG,L,tC,tS}{read nil}{
  for all (i,j) edge in tG{
    tInt ^c_i,j = t_alloc(tC * sizeof_tInt);
    tInt ^b_i,j = c_i,j; ^e_i,j = c_i,j;
    while(L_i,j != empty && L_i,j = m_s,d⌢X){
      &e_i,j = m_s,d; e_i,j++; L_i,j = X;};}
  for(k=1;k=<tn;k++){
    Traffic = Sum{length(L_i,j), for (i=k or j=k)};
    s_i = New_speed(Traffic,tS);}
}{speak  tn, tG, L1=(^c_i,j,^b_i,j,^e_i,j),
         S1=(s_i),tC,tS}{write nil}
```

```
module N{listen tn,tG,L1,S1,tC,tS,ti}{read id,next}{
  k = random in {1,2,3};
  switch(k){
  case k=1:  //send zero or one item
    m_id,d = random, with m∈Msg and d∈{1,...,tn}\{id};
    if(e_id,next(d) =< tC){
      &e_id,next(d) = m_id,d; e_id,next(d)++;
    } else {postpone sending};
  case k=2:  //receive zero or one item
    in = random, with b_in,id != e_in,id
      and &b_in,id = m_s,d and d = id;
    if(in != empty){b_in,id++};
  case k=3:  //route zero or one item
    in = random, with b_in,id != e_in,id
      and &b_in,id = m_s,d and d != id;
    if(in != empty && e_id,next(d) =< tC){
      &e_id,next(d) = m_s,d; e_id,next(d)++;
      b_in,id++;}
  }
}{speak tn,tG,L1,S1,tC,tS,ti}{write id,next}
```

```
module N2{listen tn,tG,L1,S1,tC,tS}{read nil}{
  for all (i,j) edge in tG{ L_i,j = empty;
     while(b_i,j != e_i,j){
       L_i,j = L_i,j⌢&b_i,j; b_i,j++;};}
}{speak tn,tG,L=(L_i,j),tC,tS}{write nil}
```

Table 2.11: Modules of the **Network** program

Chapter 3

Model-based testing and finite state representations for Event-B

Contributors: Ionut Dinca (UniPit), Florentin Ipate (UniPit), Raluca Lefticaru (UniPit), Michael Leuschel (UDUS), Laurentiu Mierla (UniPit), Daniel Plagge (UDUS), Alin Stefanescu (UniPit), Cristina Tudose (UniPit), and Sebastian Wieczorek (SAP)

The main topic of this research is model-based testing (MBT) for Event-B, that is test generation from Event-B models. This is an important subject that has not received much attention until now. Moreover, it is a specific requirement from the deployment partner SAP.

Among the several ideas investigated in the project, in this chapter we focus on MBT using techniques like automata-learning, meta-heuristics searches and multi-objective optimizations, described in the following three sections. The last section presents a Rodin plug-in implementing these methods. Experiments probing the efficiency of the prototype were performed on benchmarks of Event-B models from the DEPLOY repository.

The material in this chapter is based on several papers already accepted or under review: [104] used in Section 3.1, [105] used in Section 3.2, [79] and [146] used in Section 3.3, [132] used in Section 3.4, and [70] used in Section 3.5.

Note that in this chapter we only report work done by UniPit. In DEPLOY Enlarged EU, also University of Dusseldorf (UDUS) and SAP investigated MBT for Event-B (using model checking or constraint solving). We do not report this work here, because this work was already reported in other deliverables, namely Deliverable D44 (see Chapter 10) and Deliverable D42 (see Chapter 4), and the joint paper with SAP [69]. Moreover, to these deliverables contributed also UniPit, so there is a bit of overlap between these deliverables and the current one (Section 3.1 and Section 3.4). However, the rest of the sections in this chapter are new.

The research in this chapter was performed under the Task 9.10 of the DEPLOY Extended EU DoW.

3.1 Model-learning and MBT for Event-B

We propose an approach which, given a state-transition model of a system, constructs, in parallel, an *approximate automaton* model and a *test suite* for the system. The approximate model construction relies on a variant of Angluin's automata learning algorithm, adapted to finite cover automata. A *finite cover automaton* represents an approximation of the system which only considers sequences of length up to an established upper bound ℓ. Crucially, the size of the cover automaton, which normally depends on ℓ, can be significantly lower than the size of the exact automaton model. In this way, by appropriately setting the value of the upper bound ℓ, the state explosion problem normally associated with constructing and checking state based models can be addressed. The proposed approach also allows for a gradual construction of the model and of the associated test suite, with complexity and time savings, but also with improvements in the accurateness of the obtained models and tests. The approach is presented and implemented in the context of the Event-B modeling language, but its underlying ideas and principles are much more general and can be applied to any system those behavior can be suitably described by a state-transition model.

The concept of state is at the heart of model-based testing and many test generation techniques from finite state machines (FSMs) exist. However, FSMs are not powerful enough to efficiently model realistic systems and so extended finite state machines (EFSMs), such as Statecharts, are used instead; these combine a FSM-like control with suitable data variables and operations for these variables, to offer an intuitive, yet rigorous means for system modeling and analysis. Testing from an EFSM usually involves transforming the EFSM into an equivalent FSM (whose states are given by the state-variable value combinations of the original EFSM) and then applying FSM-based test generation techniques. However, for many systems, the equivalent FSM may have many more states than the length of the tests that can realistically be performed, or, furthermore, the number of states of the resulting FSM may be so large that it is impossible to even construct it. This is the well-known state explosion problem. Despite the existence of numerous techniques for alleviating this problem, state explosion remains one of the major obstacles for efficient model-based test generation from state-based models.

In this section we propose an approach which, given a state-transition model (EFSM) of a system, constructs, in parallel, an *approximate FSM* model and a *test suite* for the system. The (approximate) model construction relies on a variant of Angluin's automata learning algorithm [74], adapted to finite cover automata [106]. A *finite cover automaton* [76] of a finite set L is a finite automaton which accepts all sequences in L but may also accept sequences that are longer than every sequence in L. The main advantage of a finite cover automaton is that its size (number of states), which normally depends on ℓ, can be significantly lower than the size of the automaton which accepts exactly the language L [76]. In practice, an upper bound ℓ on the length of the considered sequences will be established and the constructed model will have to conform to the original model for all sequences of length at most ℓ. In this way, by appropriately setting the value of the bound ℓ, the state explosion problem normally associated with constructing and checking state based models can be addressed. Furthermore, test generation from finite cover automata fits very well with common testing practices, that usually require test cases of short to medium length and can also be regarded as a natural complement to Bounded Model Checking (BMC) based on SAT methods [77], which is gaining popularity in the formal verification community.

The proposed approach also allows for a gradual construction of the model and of the associated test suite: the FSM model and test suite for an initial version of the system are reused in the construction of a more elaborated and complex version, with complexity and time savings, but also with improvements in the accurateness of the obtained models and tests.

The approach is presented and implemented for the Event-B modeling language, but its underlying ideas and principles are much more general and can be applied to any system those behavior can be suitably described by a state-transition model. As Event-B does not even distinguish between state and data variables, such models are very suitable means for evaluating our approach.

Given an Event-B model and an upper bound ℓ, the proposed approach will incrementally construct finite cover automata that will eventually accept all executable sequences of length less than or equal to ℓ. As a by-product of the automata learning algorithm, a *set of test cases* associated with the cover automata is also maintained and evolved during the iterations. This test suite can be used for conformance testing of the modeled system. The test cases in the test suite are provided together with the associated *test data* that makes them executable on the Event-B model.

The contributions of our approach are threefold:

– first, it constructs a successive set of finite approximation models for the set of Event-B executable traces up to a length ℓ. The construction exploits the restriction given by the bound ℓ to obtain models of reduced size (number of states) compared to exact automaton models. Moreover, the cover automata are minimal by construction.

– second, in parallel with automata construction, we incrementally generate conformance test suites for the investigated Event-B models. By construction, the generated test cases satisfy certain minimality properties regarding their lengths. This fits very well with the testing practice that usually requires short test cases.

– third, the Event-B method deploys model refinement as a means to handle modeling complexity. The two contributions above can be applied incrementally, allowing the reuse of the learned model and test cases from the abstract to the more concrete levels.

The section is structured as follows. Subsection 3.1.1 recalls theoretical foundations, including the used algorithm for the cover automata. Subsections 3.1.3 and 3.1.5 describe the adaptation to Event-B and its implementation, respectively. Subsection 3.1.7 concludes the section.

3.1.1 Theoretical background

This subsection, which is largely adapted from our previous work [106], presents the L^ℓ algorithm and its automata-related concepts.

Before continuing, we introduce the notations used in the section. For a finite alphabet A, A^* denotes the set of all finite sequences with members in A. ϵ denotes the empty sequence. For a sequence $a \in A^*$, $\|a\|$ denotes the length (number of symbols) of a; in particular $\|\epsilon\| = 0$. For a finite set of sequences $U \subseteq A^*$, $\|U\|$ denotes the length of the longest sequence(s) in U. For $a, b \in A^*$, ab denotes the concatenation of sequences a and b. a^n is defined by $a^0 = \epsilon$ and $a^n = a^{n-1}a$, $n \geq 1$. For $U, V \subseteq A^*$, $UV = \{ab \mid a \in U, b \in V\}$; U^n is defined by $U^0 = \{\epsilon\}$ and $U^n = U^{n-1}U$, $n \geq 1$. $A[n] = \bigcup_{0 \leq i \leq n} A^i$ denotes the sets of sequences of length less than or equal to n with members in the alphabet A. For a sequence $a \in A^*$, $b \in A^*$ is said to be a *prefix* of a if there exists a sequence $c \in A^*$ such that $a = bc$. The set of all prefixes of a is denoted by $pref(a)$; for $U \subseteq A^*$, $pref(U) = \bigcup_{a \in U} pref(a)$. For a sequence $a \in A^*$, $b \in A^*$ is said to be a *suffix* of a if there exists a sequence $c \in A^*$ such that $a = cb$. For a finite set A, $card(A)$ denotes the number of elements in A.

Finite automata - general concepts

We start by introducing some classic definitions from automata theory.

A *deterministic finite automaton* (*DFA*) M is a tuple (A, Q, q_0, F, h), where: A is the finite *input alphabet*; Q is the finite *set of states*; $q_0 \in Q$ is the *initial state*; $F \subseteq Q$ is the *set of final states*; h is the *next-state*, $h : Q \times A \longrightarrow Q$. A DFA is usually described by a *state-transition diagram*.

The next-state function h can be naturally extended to a function $h : Q \times A^* \longrightarrow Q$. A state $q \in Q$ is called *reachable* if there exists $s \in A^*$ such that $h(q_0, s) = q$. M is called *reachable* if all states of M are reachable.

Given $q \in Q$, the set L_M^q is defined by $L_M^q = \{s \in A^* \mid h(q, s) \in F\}$. When q is the initial state of M, the set is called the *language accepted by* M and the simpler notation L_M is used. Given $Y \subseteq A^*$, two states $q_1, q_2 \in Q$ are called *Y-equivalent* if $L_M^{q_1} \cap Y = L_M^{q_2} \cap Y$. Otherwise q_1 and q_2 are called *Y-distinguishable*. If $Y = A^*$ then q_1 and q_2 are simply called *equivalent* or *distinguishable*, respectively. Two DFAs are called (Y-)equivalent or (Y-)distinguishable if their initial states are (Y-)equivalent or (Y-)distinguishable, respectively.

A DFA M is called *reduced* if every two distinct states of M are distinguishable. A DFA M is called *minimal* if any DFA that accepts L_M has at least the same number of states as M. A DFA M is minimal if and only if M is reachable and reduced. Furthermore, there is an unique (up to a renaming of the state space) minimal DFA that accepts a given regular language.

Now let us also introduce the concept of *deterministic finite cover automaton* (DFCA). Informally, a DFCA of a finite language U, as defined by Câmpeanu et al. [76], is a DFA that accepts all sequences in U and possibly other sequences that are longer than any sequence in U.

In this section we use a slightly more general concept, as defined in [106]: given a finite language $U \subseteq A^*$ and a positive integer ℓ that is greater than or equal to the length of the longest sequence(s) in U, a *deterministic finite cover automaton* (*DFCA*) of U w.r.t. ℓ is a DFA M that accepts all sequences in U and possibly other sequences that are longer than ℓ, i.e. $L_M \cap A[\ell] = U$, where $A[\ell] := \bigcup_{0 \leq i \leq \ell} A^i$. A DFCA M of U w.r.t. ℓ is called *minimal* if any DFCA of U w.r.t ℓ has at least the same number of states as M. Note that, unlike the case in which the acceptance of the exact language is required, the minimal DFCA is not necessarily unique (up to a renaming of the state space) [106].

Naturally, a DFA that accepts a finite language U is also a DFCA of U w.r.t. any $\ell \geq \|U\|$. Consequently, the number of states of a minimal DFCA of U w.r.t. ℓ will not exceed the number of states of the minimal DFA accepting U. Furthermore (and more importantly from the point of view of practical applications), the size of a minimal DFCA of U w.r.t. ℓ can be much smaller than the size of the minimal DFA that accepts U [106].

The L^ℓ algorithm for learning finite cover automata

Learning regular languages from queries was introduced by Angluin in [74]; the paper also provides a learning algorithm, called L^*. The L^* algorithm infers a regular language, in the form of a DFA from the answers to a finite set of membership queries and equivalence queries. A *membership query* asks whether a certain input sequence is accepted by the system under test or not. In addition to membership queries, L^* uses *equivalence queries* to check whether the learning algorithm is completed.

In a recent paper [106], we extended Angluin's work by proposing an algorithm, called L^ℓ, for learning a DFCA. Given an unknown finite set U and a known integer ℓ that is greater than or equal to the length of the longest sequence(s) in U, the L^ℓ algorithm will construct a minimal DFCA of U w.r.t. ℓ. Analogously to L^*, the L^ℓ algorithm uses membership and language equivalence queries to find the automaton in polynomial time.

The L^ℓ algorithm construct two sets: S, a non-empty, prefix-closed set of sequences and W, a non-empty, suffix-closed set of sequences. Additionally, S will not contain sequences longer than ℓ and W will not contain sequences longer than $\ell - 1$, i.e. $S \subseteq A[\ell]$ and $W \subseteq A[\ell - 1]$.

The algorithm keeps an *observation table*, which is a mapping T from a set of finite sequences to $\{0, 1, -1\}$. The sequences in the table are formed by concatenating each sequence of length at most ℓ from the set $S \cup SA$ with each sequence from the set W. Thus, the table can be represented by a two-dimensional array with rows labeled by elements of $(S \cup SA) \cap A[\ell]$ and columns labeled by elements of W.

The function $T : ((S \cup SA) \cap A[\ell])W \longrightarrow \{0, 1, -1\}$ is defined by $T(u) = 1$ if $u \in U$, $T(u) = 0$ if $u \in A[\ell] \setminus U$ and $T(u) = -1$ if $u \notin A[\ell]$. The values 0 and 1, respectively, are used to indicate whether a sequence is contained in U or not. However, only sequences of length less than or equal to ℓ are of interest. For the others, an extra value, -1, is used.

In order to compare the rows in the observation table, a relation on these rows, called *similarity*, is used. We say that rows s and t are *k-similar*, $1 \leq k \leq \ell$, and write $s \sim_k t$ if, for every $w \in W$ with $\|w\| \leq k - max\{\|s\|, \|t\|\}$, $T(sw) = T(tw)$. Otherwise, s and t are said to be *k-dissimilar*, written $s \not\sim_k t$. In other words, the table values of rows s and t must coincide for every column w for which the lengths of sw and tw are both less than or equal to k. The relation $\sim_k$ is not an equivalence relation since it is not transitive [106]. When $k = \ell$, we simply say that s and t are *similar* or *dissimilar* and write $s \sim t$ or $s \not\sim t$, respectively. It can be observed that similarity of rows s and t requires all corresponding non-negative values of the two rows to coincide.

Using the similarity relation, two properties of an observation table are defined: consistency and closedness.

The observation table is *consistent* if, for every k, $1 \leq k \leq \ell$, whenever rows $s_1 \in S$ and $s_2 \in S$ are k-similar, rows $s_1 a$ and $s_2 a$ are also k-similar for all $a \in A$.

The observation table is *closed* if, for all rows $s \in SA$, there exists row $t \in S$ with $\|t\| \leq \|s\|$, such that $s \sim t$.

Consider, for example, $A = \{a, b\}$, $\ell = 3$ and Table 3.1 (left hand side) - in which a double horizontal line is used to separate the rows labeled with elements of S from the rows labeled with elements of $SA \setminus S$ - to be the current observation table ($S = \{\epsilon, a, b, aa, bb\}$, $W = \{\epsilon, a\}$). The observation table is not consistent since, for $k = 2$, $s_1 = \epsilon, s_2 = b$, $w = \epsilon$ and $\alpha = b$ satisfy $s_1 \sim_k s_2$, but $T(s_1 \alpha w) \neq T(s_2 \alpha w)$. On the other hand, the observation table is closed.

T	ϵ	a
ϵ	0	0
a	0	1
b	0	0
aa	1	0
bb	1	0
ab	0	0
ba	0	0
aaa	0	-1
aab	0	-1
bba	0	-1
bbb	0	-1

T	ϵ	a	b
ϵ	0	0	0
a	0	1	0
b	0	0	1
aa	1	0	0
bb	1	0	0
ab	0	0	0
ba	0	0	1
aaa	0	-1	-1
aab	0	-1	-1
bba	0	-1	-1
bbb	0	-1	-1

Table 3.1: Observation table of running example (left table) and its updated form that is consistent and closed (right table)

The algorithm starts with $S = W = \{\epsilon\}$. It periodically checks the consistency and closedness properties and extends the table accordingly. When both conditions are met, the DFA $M(S, W, T)$ corresponding to the table is constructed (details will be provided later on) and it is checked whether the language L accepted by $M(S, W, T)$ satisfies $L \cap A[\ell] = U$ (this is called a "language query"). If the language query fails, a counterexample t is produced, the table is expanded to include t and all its prefixes and the consistency and closedness checks are performed once more. Eventually, the language query will succeed and the algorithm will return a minimal DFCA of U w.r.t. ℓ.

Since in our approach we will separate the construction of the observation table and of the corresponding DFCA (which is the actual processing performed by the algorithm) from the language queries (which represent the user intervention), only the processing performed between two language queries is presented in pseudo-code in Fig. 3.1 (in what follows this will be referred to as the LearnDFCA procedure).

The *LearnDFCA* procedure starts with the current values of S, W and the current observation table T. It periodically checks whether the consistency and closedness properties are violated and extends the table by adding a new row or a new column to the table, respectively:

– In order to check consistency, the procedure will search for $w \in W$ and $a \in A$ such that aw will distinguish between two rows s_1 and s_2 that are not distinguished by any sequences in W of length less than or equal to aw; in order to find the shortest such sequence aw, the search will be performed in increasing order of length of w. The search is repeated until all elements of W have been processed; as these are processed in increasing order of their length, any sequence aw that has been added to W as a result of an incorrect

```
Procedure LearnDFCA

Input: $S$, $W$ and the current observation table $T$.
Repeat
    \— Check consistency —\
    For every $w \in W$, in increasing order of $\|w\| = i$ do
        Search for $s_1, s_2 \in S$ with $\|s_1\|, \|s_2\| \leq \ell - i - 1$
        and $a \in A$ such that $s_1 \sim_k s_2$, where
        $k = max\{\|s_1\|, \|s_2\|\} + i + 1$, and $T(s_1aw) \neq T(s_2aw)$.
        If found then
            Add $aw$ to $W$.
            Extend $T$ to $(S \cup SA)W$ using membership queries.
    \— Check closedness —\
    Set $new\_row\_added = false$.
    Repeat for every $s \in S$, in increasing order of $\|s\|$
        Search $a \in A$ such that $sa \not\sim t\ \forall t \in S$ with $\|t\| \leq \|sa\|$.
        If found then
            Add $sa$ to $S$.
            Extend $T$ to $(S \cup SA)W$ using membership queries.
            Set $new\_row\_added = true$.
    Until $new\_row\_added$ or all elements of $S$ were processed
Until $\neg new\_row\_added$
Construct $M(S, W, T)$.
Return $M(S, W, T)$.
```

Figure 3.1: The learning procedure *LearnDFCA*

consistency check will be itself processed in the same “For” loop.

– In order to check closedness, the procedure will search for $s \in S$ and $a \in A$ such that sa is dissimilar to any of the current rows t for which $\|t\| \leq \|sa\|$; similarly, the search is performed in increasing order of length of s. If such s and a are found, then sa is added to the observation table and the algorithm will check again its consistency.

Consider once again Table 3.1 (left hand side) as the current observation table. This will fail the consistency check for $i = 0$ and $k = 2$: $s_1 = \epsilon, s_2 = b$, $w = \epsilon$ and $\alpha = b$ satisfy $s_1 \sim_k s_2$, but $T(s_1\alpha w) \neq T(s_2\alpha w)$. Consequently, $\alpha w = b$ is added to W and so Table 3.1 (right hand side) is the resulting observation table. This is both consistent and closed and so the DFA $M(S, W, T)$ is constructed.

The state set of DFA $M(S, W, T)$ is formed by taking all minimum, mutually dissimilar sequences from S, where the minimum is taken according to the quasi-lexicographical order on A^* [106]. For Table 3.1 (right hand side), this is $Q = \{\epsilon, a, b, aa\}$ (since $bb \sim aa$) and the corresponding DFA is as represented in Fig. 3.2 (in which final states are drawn in double line, whereas non-final states are drawn in single line; the initial state is $q0$). The formal definition of $M(S, W, T)$ is given in [106], for simplicity this is not reproduced here. Further details regarding the L^ℓ algorithm, including proofs of correctness and termination and examples which illustrate its functioning, can also be found in [106].

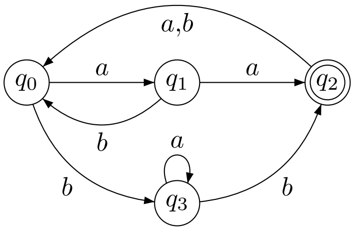


Figure 3.2: The DFA corresponding to Table 3.1 (right hand side)

3.1.2 Black-box testing for finite cover automata

Before proceeding, we briefly outline the W-method for bounded sequences, which we proposed in [103]. This is not central to our approach, but may be used for answering language queries, as discussed later.

Given a DFCA model $M = (A, Q, q_0, F, h)$ of a system and an upper bound ℓ, this method generates a set of sequences to check if the implementation under test, which is modeled by an unknown finite automaton I, behaves as defined by M for all sequences of length at most ℓ. In other words, if the languages accepted by M and I are L_M and L_I, respectively, the W-method will construct a finite set of sequences $X \subseteq A[\ell]$ such that $L_M \cap X = L_I \cap X$ implies $L_M \cap A[\ell] = L_I \cap A[\ell]$.

The implementation is a black box and so, naturally, I is not known; however, it is assumed that the maximum number of states of I can be estimated; the difference between this estimated maximum and the number of states of M is denoted by k (if the difference is negative then we take $k = 0$).

Naturally, one can always take X to be the set of all sequences of length up to ℓ; however, the W-method produces a much smaller set, whose size is polynomial in the number of states of M (but exponential in k). The construction of X is based on two sets: a *proper state cover* S and a *strong characterization set* W of M. S is called a proper state cover of M if it contains sequences of minimum length that reach all states of M, i.e. for every $q \in Q$ there exists $s \in S$ such that: $h(q_0, s) = q$ and $\|s\| \leq \|t\|$ for every $t \in A^*$ for which $h(q_0, t) = q$. W is called a strong characterization set if it contains sequences of minimum length that distinguish between any pair of states of the DFCA, i.e. for every $q_1, q_2 \in Q$, $q_1 \neq q_2$ there exists $w \in W$ such that: w distinguishes between q_1 and q_2 and $\|s\| \leq \|t\|$ for every $t \in A^*$ which distinguishes between q_1 and q_2. Then, for an estimated value of k, the test set has the form $X_k = SA[k+1]W \cap A[\ell]$. [1]

The W-method for bounded sequences [103], is a non-trivial generalization of the W-method for checking functional equivalence (also called the Vasilevski-Chow method). [2]

3.1.3 Model learning and test generation

We are now ready to present how we can apply the above theoretical harness. We first describe the Event-B modeling environment. Then we present our approach for incremental cover automata learning and test generation for Event-B models. The notion of refinement is also discussed at the end of the section.

Usually, in an Event-B model, all states which can be reached by feasible sequences of events are considered to be *final* states. Obviously, in this case only one non-final state is sufficient - a "sink" state which collects all infeasible paths. Although this is the situation we have encountered in all applications considered in this section, our approach is in no way restricted to this particular case. Furthermore, it would also be possible to differentiate between reachable and final configurations in an Event-B model by using a logical predicate on the global variables, i.e. a state is considered final only if the predicate holds.

Given an Event-B model, *a test case* can be defined as a sequence of events. This can be either positive, if it corresponds to an executable (feasible) path through the Event-B model, or negative, otherwise. The executability of a test case implies the existence of appropriate test data for the events, i.e. appropriate values for the local parameters that ensure that the guard of the event is true. Finally, a *test suite* is by definition a collection of test cases.

Incremental model learning

We will apply now the cover automata learning method of Section 3.1.1 to the Event-B framework. The input elements for the procedure were a finite language U and a bound ℓ. For an Event-B model, U will be the set of all executable event

[1] The model M is assumed to be a minimal DFCA of $L_M \cap A[\ell]$ w.r.t ℓ (if not, it is minimized before the method is applied), so both a proper state cover and a strong characterization set exist.

[2] [103] gives the results for Mealy machines, whereas here we adapted them for finite state machine acceptors.

```
**Procedure** IterativeConstructionDFCA

**Input**: $S_0$, $W_0$
Set $S = S_0$ and $W = W_0$
Construct $T$ for $(S \cup SA)W$
LearnDFCA
**While** the constructed automaton $M(S, W, T)$ is not correct **do**
    Provide a counterexample $w$
    Add $w$ and all its prefixes to $S$
    Extend the observation table $T$ to $(S \cup SA)W$
    LearnDFCA
Minimize $S$ and $W$
**Output**: $M(S, W, T)$, (minimized) $S$ and $W$, observation table
and the corresponding test sequences
```

Figure 3.3: The iterative procedure of constructing the DFCA

sequences of length maximum ℓ. The alphabet of U is the set of events in the model, which we denote by A.

Given the above U and ℓ, our approach gradually constructs both (1) a DFCA for the Event-B model and (2) an associated test set. The test set will be constructed using information from the observation table (paths through the model) and the actual test data to drive the executable paths. In this subsection we discuss the model learning cycle and in the following two subsections the test suite creation.

The proposed procedure consists of a number of steps; at each step, a new DFCA and test set is produced. The outline of this procedure is depicted in Fig. 3.3. Unlike the original L^ℓ algorithm, the procedure does not start with empty S and W, but with some initial values S_0 and W_0, which reflect the current knowledge about the DFCA model.

In case S_0 and W_0 are carefully chosen by a human with a good insight in the model, the constructed $M(S, W, T)$ will be close to the correct DFCA and so the "while"-loop will be executed fewer times or not at all, saving in this way computational resources. At limit, when either S_0 or W_0 are correctly chosen from the outset, the constructed $M(S, W, T)$ will be correct and the "while"-loop will not be executed at all. (In Subsection 3.1.6 we show that a proper state cover of the Event-B model is such a "correctly chosen" S_0 and a strong characterization set is a "correctly chosen" W_0.)

Otherwise, whenever the DFCA is found to be inaccurate, a counterexample (i.e. a sequence s with $\|s\| \leq \ell$ such that $s \in U$ but s is not accepted by the DFCA or vice versa) must be found and the observation table should be extended accordingly. Practical modalities for finding such a counterexample will discussed later in this section.

Therefore, two main cases can be distinguished:

- **Case 1:** The procedure is executed for the first time. In this case, the initial sets S_0 and W_0 are based on an initial estimation of the states of the model. In the worst case (when no initial estimation is available), we take $S_0 = \{\epsilon\}$, $W_0 = \{\epsilon\} \cup A$. (As it emerged from our empirical evaluation (Section 3.5.3), in many cases states can be distinguished by singleton sequences, and so initially we consider W to contain all event names, i.e. A).

- **Case 2:** The procedure has already been applied at least once and, consequently, a DFCA model exists. Suppose that this model is not totally accurate and needs to be improved. This may happen for a number of reasons:

  - **Subcase 1:** the Event-B model has been modified or augmented due to changes in the requirements.
  - **Subcase 2:** the Event-B model has not been changed but the associated DFCA is deemed to be insufficient for testing purposes. In this case, the upper bound ℓ is increased according to the existing testing needs and the procedure is executed once more for the new value of ℓ.
  - **Subcase 3:** the existing Event-B model has been refined and extra detail has been added (using the Event-B refinement). This subcase will be discussed later in subsection 3.1.4.

  In this (second) case, S_0 and W_0 are the values of S and W from the previous iteration. In fact, it is not necessary to reuse the entire sets S and W. As shown in Subsection 3.1.6, It is sufficient to extract from them two minimal subsets $S_{min} \subseteq S$ and $W_{min} \subseteq W$, where S_{min} is the set of all minimum, mutually dissimilar sequences from S, and W_{min} is the set of minimum sequences from W which distinguish between any two dissimilar sequences of S (the formal definitions are given in Subsection 3.1.6) - this corresponds to the minimization step in Fig. 3.3.[3] The construction of S_{min} and W_{min} is not computationally expensive; both these subsets are selected by simply scanning the observation table, so the complexity is linear in its size. Additionally, S_{min} is actually the state set of $M(S, W, T)$, so it is computed by

[3] Intuitively, this is because S_{min} and W_{min} still remain a proper state cover and a strong characterization set of $M(S, W, T)$, so the language equivalence (modulo the upper bound ℓ) against any other automaton with the same number of states is ensured.

> the algorithm anyway. The observation table T (corresponding to the minimized S and W) is also partially/totally re-constructed in the next iteration as follows. In the first subcase, since the Event-B model has been changed, the value of T must be re-checked for all sequences in $(S \cup SA)W \cap A[\ell]$. In the second subcase, only the sequences in $(S \cup SA)W$ whose length is greater than the previous ℓ need to be re-processed.

Note that (Case 2 / Subcase 1) can also be applied even if the procedure has not been applied before but an automaton model of the system exists from other sources (e.g. has been developed during the design phase), but has become obsolete. In this case, S_{min} and W_{min} can also derived from the existing model, as explained above, so the information contained in the existing model is reused in the construction of the new model,

Example. We illustrate the iterative process of constructing the DFCAs with a system for controlling the cars on a narrow bridge between an island and the mainland. This example is particularly relevant since it is used to introduce the main Event-B concepts in Abrial's textbook [94].

The modeled system is equipped with two traffic lights with two colors: green and red. The traffic lights control the entrance to the bridge at both ends. Cars are not supposed to pass on a red traffic light, only on a green one. There are also some car sensors situated at both ends of the bridge which are used to detect the presence of a car entering or leaving the bridge. The system has two main additional constraints: the number of cars on the bridge and island is limited and the bridge is one-way.

We present here only the first two levels of refinement (see Fig. 3.4 and Fig. 3.5). The first model M_0 is very simple. The events ML_out and ML_in correspond to cars entering and leaving the island-bridge compound, respectively. The context contains a single constant d, which is a natural number denoting the maximum number of cars allowed to be on the island-bridge compound at the same time. The single variable n of the machine M_0 denotes the actual number of cars.

In the first refinement, the machine M_1 introduces the bridge. The events ML_out and ML_in correspond now to cars leaving the mainland and entering the bridge or leaving the bridge and entering the mainland, respectively. In addition, the events IL_in and IL_out correspond to cars entering and leaving the island, respectively. The variable n *is now replaced* by three variables: a (the number of cars on the bridge and going to the island), b (the number of cars on the island) and c (the number of cars on the bridge and going to the mainland).

Finally, the second refinement introduces the two traffic lights, named ml_tl and il_tl. The model M_2 has two new events to turn the value of the traffic lights color to green when they are red: ML_tl_green and IL_tl_green. In order to make the colors change in a more disciplined way, two more variables ml_pass

```
Machine $M_0$:
  Variables:  $n$
  Event INITIALISATION $\widehat{=}$ begin $n := 0$ end
  Event $ML\_out$ $\widehat{=}$ when $n < d$ then $n := n + 1$ end
  Event $ML\_in$ $\widehat{=}$ when $n > 0$ then $n := n - 1$ end
```

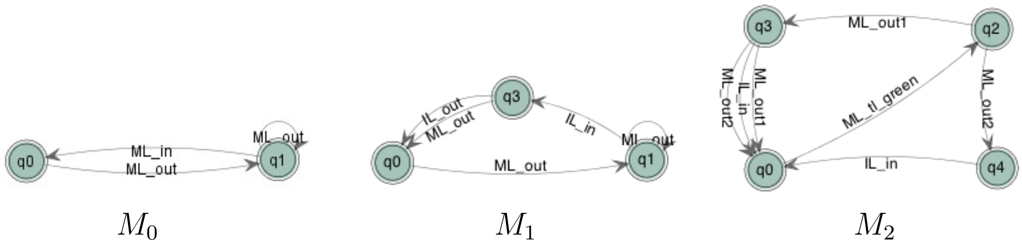
```
Machine $M_1$ refines $M_0$ :
  Variables:  $a, b, c$
  Event INITIALISATION $\widehat{=}$ begin $a, b, c := 0, 0, 0$ end
  Event $ML\_out$ refines $ML\_out$ $\widehat{=}$ when $a + b < d \ \wedge \ c = 0$ then $a := a + 1$ end
  Event $ML\_in$ refines $ML\_in$ $\widehat{=}$ when $c > 0$ then $c := c - 1$ end
  Event $IL\_in$ $\widehat{=}$ when $a > 0$ then $a, b := a - 1, b + 1$ end
  Event $IL\_out$ $\widehat{=}$ when $0 < b \ \wedge \ a = 0$ then $b, c := b - 1, c + 1$ end
```

```
Machine $M_2$ refines $M_1$ :
  Variables:  $a, b, c, ml\_tl, il\_tl, il\_pass, ml\_pass$
  Event INITIALISATION $\widehat{=}$begin $(a, b, c := 0, 0, 0)$,
                                       $(ml\_tl, il\_tl := red, red)$,
                                       $(il\_pass, ml\_pass := 1, 1)$
                                 end
  Event $ML\_out1$ refines $ML\_out$ $\widehat{=}$when $ml\_tl = green \ \wedge \ a + b + 1 < d$
                                      then $a, ml\_pas := a + 1, 1$ end
  Event $ML\_out2$ refines $ML\_out$ $\widehat{=}$when $ml\_tl = green \ \wedge \ a + b + 1 = d$
                                      then $(a, ml\_pas := a + 1, 1), ml\_tl := red$ end
  Event $IL\_out1$ refines $IL\_out$ $\widehat{=}$when $il\_tl = green \ \wedge \ b > 1$
                                      then $(b, c := b - 1, c + 1), il\_pas := 1$ end
  Event $IL\_out2$ refines $IL\_out$ $\widehat{=}$when $il\_tl = green \ \wedge \ b = 1$
                                      then $(b, c := b - 1, c + 1), (il\_tl, il\_pas := red, 1)$
                                      end
  Event $ML\_tl\_green$ $\widehat{=}$when $ml\_tl = red \ \wedge \ a + b < d \ \wedge \ c = 0 \ \wedge \ il\_pass = 1$
                       then $(ml\_tl, il\_tl := green, red), ml\_pass := 0$ end
  Event $IL\_tl\_green$ $\widehat{=}$when $il\_tl = red \ \wedge \ 0 < b \ \wedge \ a = 0 \ \wedge \ ml\_pass = 1$
                       then $(ml\_tl, il\_tl := red, green), il\_pass := 0$ end
  Event $IL\_in$ refines $IL\_in$ $\widehat{=}$ when $a > 0$ then $a, b := a - 1, b + 1$ end
  Event $ML\_in$ refines $ML\_in$ $\widehat{=}$ when $c > 0$ then $c := c - 1$ end
```

Figure 3.4: The first two refinements of the "Cars on the bridge" example (from Abrial [94])

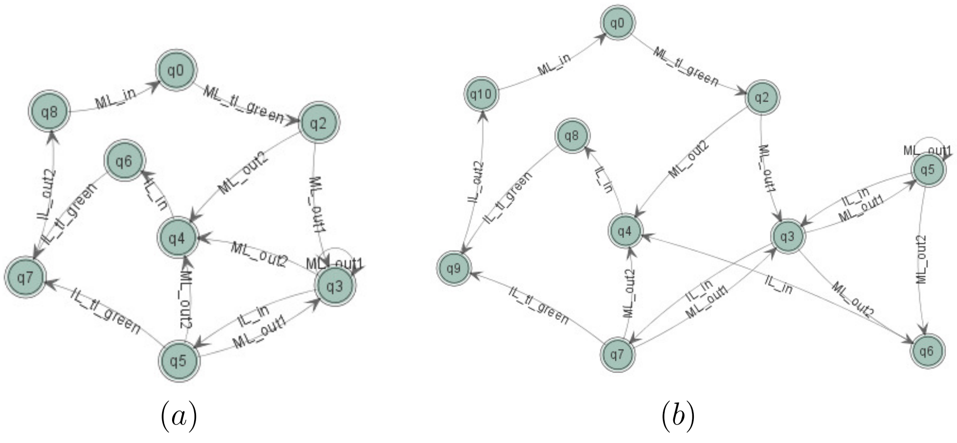
and il_pass are introduced: ml_pass = *TRUE* signifies that at least one car has passed the bridge going to the island since the mainland traffic light last turned green; similarly for il_pass = *TRUE*.

First, we start the learning process with $\ell = 3$ (*Case 1*); for each of the three models, the procedure is executed with initial values $S_0 = \{\epsilon\}$ and W_0 equal to



M_0 M_1 M_2

Figure 3.5: DFCAs for CarsOnBridge given $\ell = 3$



(a) (b)

Figure 3.6: DFCAs for (a) M_2 and $\ell = 6$; and (b) the DFCA improvement after providing a counterexample.

the corresponding input alphabet plus the empty sequence; the resulted DFCAs (plotted using our plug-in) are presented on the right hand side of Fig. 3.4 (for simplicity, the sink states are not shown).

Suppose now that we want to improve the DFCA for M_2 by increasing the upper bound ℓ (*Case 2 / Subcase 2*). For $\ell = 6$, we obtain the DFCA in Fig. 3.6(a), which has more states and transitions (and covers events like ML_in that were not covered for $\ell = 3$).

In order to illustrate the iterative DFCA construction (cf. Fig. 3.3), we provide a counterexample path for the current DFCA associated to M_2. For instance, the following sequence of length 5:

$$w = ML\_tl\_green,\ ML\_out1,\ ML\_out2,\ IL\_in,\ IL\_in$$

is feasible in M_2, but it is not accepted by the DFCA in Fig. 3.6(a). The new DFCA taking into account the counterexample w (see the while-loop of Fig. 3.3) is presented in Fig. 3.6(b). It can be observed that the path $q_0 \rightarrow q_2 \rightarrow q_3 \rightarrow q_6 \rightarrow q_4 \rightarrow q_8$ in the new DFCA corresponds to the path w in M_2.

Naturally, finding a counterexample is the most problematic part of our approach and a model checker such as ProB can be of great assistance. There are several possibilities to do this:

- interactively, using the experience of the human testers that have a good understanding of the model: Testers can use the simulation and animation capabilities of ProB to discover counterexamples, that are fed to the learning algorithm. Moreover, high-priority scenarios that the testers deem as important can be introduced into the learning loop and the associated tests will be covered by the DFCA.

- by testing language equivalence, using the W-method for bounded sequences outlined in section 3.1.2: Recall that, given a DFCA model M of a system and an upper bound ℓ, this method generates a test set to check if the implementation under test, modeled by an unknown automaton I, behaves as defined by M for all sequences of length at most ℓ. The test set has the form $X_k = SA[k+1]W \cap A[\ell]$, where S and W are a proper state cover and a strong characterization set of M, respectively, and k is the difference between the estimated maximum number of states of I and the number of states of M. In our case, the model M corresponds to the current DFCA $M(S, W, T)$ and the implementation under test to the Event-B model (more precisely, an approximation of the Event-B model, which contains all set of executable paths of length up to ℓ). Now, the sets S and W in the observation table satisfy the definitions of a proper state cover and a strong characterization set of $M(S, W, T)$, respectively. Thus, for $k = 0$, the test set X_0

is actually the set of sequences in the observation table, so, if the Event-B model is known to have no more states than the current DFCA, this step is already completed. Otherwise, testing the behavioral equivalence between the current DFCA and the Event-B model corresponds to gradually increasing k until a counterexample is found (a test case produces a different result on the Event-B model compared to the DFCA) or we are satisfied that the DFCA is correct. Note that the size of the test set is exponential in k and so using the W-method for a large k may be expensive.

– by encoding the language equivalence problem into the ProB model checker: For instance the complement of the DFCA is encoded into a CSP process P and ProB will try to run the Event-B machine and P in parallel to find a path that is accepted by Event-B but not by the DFCA. Note, however, that this procedure might be computationally expensive.

From the three options above, in the current version of our implementation we only consider the first one, in which the counterexample is manually provided (which is in fact in the spirit of the original Angluin's algorithm). This also fits well with common practice, in which human knowledge is used to guide model and test design.

Test data generation

In order to decide whether a given sequence s, $\|s\| \leq \ell$, is accepted or not by the DFCA (i.e., $s \in L^\ell$ or not), the procedure needs to check if s is a feasible path through the Event-B model. This is achieved by effectively constructing (or attempting to construct) test data to drive the given path. If the appropriate test data has been found, then $s \in L^\ell$; otherwise, the path is declared infeasible[4] and so $s \notin L^\ell$. Therefore, deciding whether $s \in L^\ell$ or not reduces to finding test data to execute the corresponding path of the Event-B model.

Then, all it remains to specify are the method(s) used to find test data to execute a given path of an Event-B model. So far two such approaches have been proposed and implemented. The first used symbolic execution and reduces this problem to solving a set of constraints [131]. The second reduces the problem to an optimization problem, which is then solved using search-based techniques (genetic algorithms) [79]. Note that the test data generation problem may be complex even for one path, when the guards are complex and the test data domain are large. In particular, the set-theoretic nature of Event-B increases the search

[4] Note that here *infeasible* means only that our tools could not find test data within reasonable time (e.g. 20 seconds for one path) and we stop searching, whereas in reality there might exists such test data. However, since we are working with approximated models, this incompleteness aspect is not very important.

space because free set variables v that are subsets of a given carrier set V (i.e. $v \subseteq V$) can take exponentially many values, $2^{card(V)}$. Consequently, most of the time taken by the execution of the procedure is spent on generating the actual test data.

3.1.4 Test suite construction

When the process of constructing the DFCA is completed, a test suite for the Event-B model has also been obtained; this is precisely the set of sequences in the observation table returned by the procedure, $X_0 = (S \cup SA)W \cap A[\ell]$ (note that the procedure returns the minimized S and W and so these are the used in the definition of X_0). The test sequences can be classified into *positive* (for which $T(x) = 1$, which correspond to feasible paths in the Event-B model) and *negative* (for which $T(x) = 0$). Naturally, test data can only be generated for feasible paths and, as explained earlier, test data generation is implicitly included in the DFCA construction procedure. Negative test sequences are also useful for testing the system implementation since they describe erroneous scenarios, which the system cannot perform in normal functioning.

Following [103], the constructed set will constitute a conformance test suite for the Event-B model modulo the bound ℓ (the ℓ-bounded behavior of the model). Such a test is more powerful that a set of tests based on state or transition coverage criteria since it covers all states and all transitions of the equivalent automaton and also checks each state and the initial and destination states of each transition. Conformance testing is especially relevant in the embedded systems domain.

Increasing ℓ, longer and more complex tests are generated. However, very complex or long test sequences are usually not the norm, so having the ability to tune the length of the test case using ℓ is an advantage of our approach. Another advantage is the fact that the method is interactive, so the tester can use its intuition to provide relevant sequences to the algorithm to learn and thus more directly influence the result of the test suite. This is in contrast to purely automatic test generation techniques that are driven by coverage criteria, where the produced tests may not be intuitive or may not cover existing standard testing scenarios in the domain. Regarding coverage criteria, if a very specific coverage criteria is sought (cf. [84]), our method can accommodate this to some extent in that the training set of sequences for the learning algorithm can be chosen according to the desired coverage. Moreover, if a simpler coverage criteria like event coverage is desired, the obtained test suite can be reduced by choosing a smaller subset that satisfies the requirement.

Relation to Event-B refinement

Very often, model design is an iterative process, in which, at each step, the existing (more abstract) model is replaced by a more concretized model through *refinement*. The incremental approach of L^ℓ allows us to reuse the learned model and test suite of the abstract model to the next more concretized model (Case 2, Subcase 3 from subsection 3.1.3), as explained below. Again, the approach is presented in the context of Event-B, but the basic ideas can be extended to other languages which provide model refinement as a way to handle complexity.

Suppose we have a refinement from *AM* (abstract model) to *CM* (concrete model). In a refinement step, new events can be introduced and the existing events can be made more concrete. Let A and A' denote the sets of events of *AM* and *CM*, respectively, and let $E \subseteq A'$ be the new events introduced in *CM* that do not refine any abstract event. Every abstract event $a \in A$ from *AM* will correspond to a set (containing one or many concrete events) from *CM*. [5] Let us denote this set *ref*(a), *ref*$(a) \subseteq A' \setminus E$. Also, let *ref*$(A) = \{$*ref*$(a) \mid a \in A\}$. Suppose S_{min} and W_{min} are the (minimized) sets produced by the application of our procedure on the abstract model *AM*. As these sets contain sequences of abstract events, they need to be transformed before they can be reused in the construction of the automaton corresponding to the concrete model *CM*. On the other hand, it can be shown that only one of the two sets (S_{min} or W_{min}) is sufficient to correctly determine the corresponding DFCA model (the other set is reconstructed by the algorithm). Furthermore, the set of all feasible paths of an Event-B model is closed under prefixing and so, naturally, a path from *AM* is transformed into a path from *CM* by gradually transforming its prefixes. Such a transformation is natural in the case of S, which is prefix-closed, but is problematic for W, which must be suffix-closed. For these reasons we choose to only transform the set S_{min}. For W, we will use the same type of heuristic as for the case in which the DFCA construction procedure is executed for the first time: W is initialized with the set A' of all events of *CM* (along with the empty sequence).

The transformation of S_{min} is given in pseudocode in Fig. 3.7; map_S denotes the mapping between each sequence in S_{min} and the corresponding sequence in the concrete model. Ultimately, the algorithm will return the prefix-closure of the image of map_S, $pref(Im(map_S))$. Y denotes the set of sequences from S_{min} that remain to be processed. Sequences are processed in increasing order of their length, so, at any time, a (the) sequence of minimum length from Y is selected. As S_{min} is prefix-closed, $map_S(x)$ is obtained by extending the transformation of its longest prefix s ($x = sa$, $s \in A^*, a \in A$). Two main cases can be distinguished.

[5] In general the correspondence may be many-to-many but in the vast majority of the models we have encountered a one-to-many correspondence is sufficient.

```
Transformation of $S$

Input: $S_{min}$ and the restriction of $T$ to $S_{min}$
$maps_S(\epsilon) = \epsilon$
$Y = S_{min} \setminus \{\epsilon\}$
While $Y \neq \emptyset$ do
    Select $x = sa$, $s \in A^*, a \in A$, a seq. in $Y$ of minimum length
    $s' = maps_S(s)$
    If $T(x) = 1$ then
      $found = find\_next(s', a, t)$
      If $found$ then
        $maps_S = maps_S \oplus (x, s't)$
        $Y = Y \setminus \{x\}$
      Else
          If $s = \epsilon$ then
            Return failure
          Else
              $Y = Y \cup (dom(maps_S) \cap \{s\}A[1])$
              $dom(maps_S) = dom(maps_S) \setminus \{s\}A[1]$
    Else
        $maps_S = maps_S \oplus (x, s'a')$ for some $a' \in ref(a)$
        $Y = Y \setminus \{x\}$
Return $S^R_{min} = pref(Im(maps_S))$
```

Figure 3.7: The transformation of S in the case of refinement

– If x is a feasible path of *AM*, then $maps_S(x)$ must also be a feasible path of *CM*. This is obtained by extending $s' = maps_S(s)$ with a sequence $t = e_1 \dots e_j a'$, where $e_1, \dots, e_j \in E$, $j \leq k$ (k is a predefined upper bound), and $a' \in$ *ref*(a). The function *find_next*(s', a, t) searches for such a t in increasing order of j; if found, the function will return *TRUE*, otherwise *FALSE*. *find_next* may be called several times with the same input parameters s and a' during the execution of the algorithm. Each time, it continues the search from where it left off, so each time a different solution is produced. If *find_next* cannot find a (new) solution, the algorithm backtracks: it removes $s \neq \epsilon$ and all sequences which extend s from the domain of $maps_S$ and adds them to Y. Consequently, the algorithm will resume by processing s. If $s = \epsilon$, the algorithm cannot backtrack any further; in this case, it stops and reports failure.

- If x is not feasible in *AM* then s' can be extended with any $a' \in$ *ref*(a). [6] Note that S_{min} contains only one such sequence since the corresponding automaton will have only one non-final ("sink") state.

In refinement, an event a from *AM* is replaced by (one or more) events *ref*(a) in *CM*, describing the system reactions in different circumstances. Furthermore, the applications of the extra events may also condition the event operation and so each application of a in *AM* is replaced by some sequence $e_1 \ldots e_j a'$ in *CM*, with $e_1, \ldots, e_j \in E$ and $a' \in$ *ref*(a).

The new events from E cannot be indefinitely enabled [94] and, furthermore, in practice it is reasonable to expect that an upper bound k on the number of times they can be applied in the absence of an event from $A' \setminus E$ can be established, and so $j \leq k$; the upper bound k is then used in the definition of the *find_next* function presented above. Thus, any feasible path $a_1 \ldots a_n$ in the abstract model can be mapped (not necessarily in an unique fashion) onto a feasible path $u_1 a'_1 u_2 a'_2 \ldots u_n a'_n$ in the concrete model, with $a'_i \in$ *ref*(a_i) and $u_i \in E[k]$, $1 \leq i \leq n$. This ensures that the transformation procedure will end successfully, so every sequence in S_{min} is refined appropriately. Finally, we need to ensure that S^R_{min} is prefix-closed and so we take the prefix-closure of the refined sequences. Note that the transformation procedure given in Fig. 3.7 is only guaranteed to terminate successfully if every feasible path in *AM* has a corresponding feasible path in *CM*. This condition is in the spirit of refinement and is satisfied by the vast majority of the applications we have encountered. However, the algorithm can be easily extended so that it terminates successfully even when not all sequences in S^R_{min} can be refined - for simplicity and due to its reduced practical value, this idea is not pursued here.

Once the set S_{min} has been transformed, the DFCA construction procedure can be executed for the concrete model *CM* with initial values $S_0 = S^R_{min}$ and $W_0 = \{\epsilon\} \cup A'$. The upper bound ℓ^R used for the concrete model *CM* also needs to be established. This will be set by the user, but, naturally, it will be greater than or equal to the length of the longest sequence in S^R_{min} plus one.

Naturally, the DFCA construction procedure can always be applied directly on the concrete model, but the strategy presented here, which reuses the information from the abstract model in the construction of the concrete model and of its associated test set, presents some key advantages:

- There is a significant number of cases (60 percent of the models considered in our experiments, see Table 3.3) for which the "reuse" strategy produces richer DFCAs than those produced "from scratch“ (i.e. by the direct application of the procedure on the concrete model); on the other hand, in

[6] Since the concrete guard is not weaker than the abstract one [94], a non-feasible path in *AM* can only give rise to non-feasible paths in *CM*.

no case in our experiments the “from scratch“ strategy produced a better model. A richer model (with a larger state space) represents a better approximation of the real system and is essential for the effectiveness of the resulting test cases. In order to obtain a more precise model, appropriate counterexamples must be supplied (and the DFCA construction procedure be run at least once more); this may add significant complexity to the test generation process.

– Even if the “reuse” strategy does not directly produce improved DFCA models, it is still preferable as it offers a way of incorporating the human knowledge into the model at the appropriate level of abstraction. Let us consider two Event-B models, *AM* (abstract) and *CM* (concrete), as above. It is likely that the first DFCA for *AM* is not satisfactory, so a richer automaton is obtained by supplying the construction procedure with appropriate counterexamples. In the “reuse” strategy, these counterexamples are propagated to the next level - they are implicitly included in the DFCA for *CM*. On the other hand, when the DFCA for *CM* is produced from scratch, all counterexamples must be produced at this level. Naturally, abstract models are simpler than concrete models, so finding counterexamples (by human intervention or automatically) for the abstract model is simpler.

3.1.5 Experimental results

In this subsection we provide the results of our experimentation on a comprehensive benchmark of Event-B models. The implementation of our algorithms was done in Java as an Eclipse plugin[7] to Rodin platform (in its latest version 2.3). The membership queries were implemented using the constraint-solving functionality of ProB and a timeout of 20 seconds per query was imposed. We run ProB with fixed internal parameters, although fine-tuning of ProB parameters may improve results in certain cases. The experiments were conducted on a Windows 7 Professional 64-bit machine with an Intel Core i7 2.80GHz (8 CPUs) processor and 12 GB of RAM.

We used a broad range of models for experimentation, including systems from the embedded systems, transportation and aerospace industries as well as academic and pedagogical Event-B models used in the literature. All the chosen 10 Event-B models are publicly available in the DEPLOY model repository[8]. Table 3.2 presents the models together with their complexity, i.e. number of refinements, number of events for each refinement and number of variables for each

[7] Installation instructions and screenshots can be found at: `http://wiki.event-b.org/index.php/MBT_plugin`

[8] `http://deploy-eprints.ecs.soton.ac.uk`

refinement. For instance, the third model is our running example "CarsOnBridge" (see Fig. 3.4). It has 3 refinements (M0/M1/M2/M3); each level (including the initial machine M0) has 3, 5, 9, and 17 events, respectively; and 1, 3, 7, and 18 variables[9], respectively. Note also that many of the models are rather complex, e.g. TrainCtrller exhibits 8 levels of refinements and the last level has 43 events and 35 variables.

Table 3.3 presents the results of our LearnDFCA procedure for the 10 models. For each model we considered for exemplification several machines (at different refinement levels) and different values of the bound ℓ. For each combination, we provide the number of states of the learned DFCA together with number of associated conformance tests and the number of iterations needed to generate the DFCA (i.e. how many times the outmost loop in Fig. 3.1 was executed). These three dimensions are given for two strategies: first one, named "from scratch" where information from the previous refinements is not used, and second, where an information reuse is taken into account as proposed in previous Section 3.1.4. As hoped, the "reuse" approach generated richer DFCAs in a smaller number of iterations.

We note that the number of produced tests may seem high, but this is because conformance testing is a strong form of testing heavily exercising the system. However, more compact test suites can be obtained according to weaker coverage criteria. For instance, test suites for state and transition coverage are readily available from the learning procedure, from the sets S and $S \cup SA$, respectively. Moreover, we have also implemented different test suite optimization algorithms that produce significantly smaller test suites.

A practical heuristic: In our experiments we have found that, in most cases, the states of the resulting automata can be distinguished by using only singleton sequences. Since these are already contained in the initial W, in these case the consistency checks performed by the DFCA are already successful and produce no effect on the observation table (the set W is not expanded). Even when there are states which can only be distinguished by sequences longer than 1, generally W is expanded much less frequently than S and so consistency checks would fail much more rarely than closedness checks. This observation led to the following heuristic: the procedure is applied first without consistency checks (at this step only S is enlarged as a consequence of the failed closedness checks) and then, once more, in which both consistency and closedness checks are performed. The heuristic should considerably reduce the overall number of consistency checks. Since consistency checks are time consuming operations (each such operation may require a large number of membership queries), the heuristic should also reduce the ex-

[9] Variables might have an integer type, but also more complex types like sets, relations or partial functions, which increase the complexity of the algorithms, especially the membership queries.

ecution time. We have implemented both variants of the procedure (the original version and one which uses the aforementioned heuristic). The experimental results shown in Table 3.4 indeed show that the heuristic produced improved results in the majority of cases, but more significantly, that the improvements are significant (over 50%) for large execution times (over 1,000 s).

Table 3.2: The complexity dimensions of the ten subjects (number of refinements, number of events and number of variables)

Subject	# of refin's	# of events per refin.	# of variables per refin.
A2A	12	4/5/5/7/9/9/12/14/15/ 16/16/17/17	2/2/4/8/8/10/11/12/13/ 15/16/18/17
BepiColombo	3	6/11/13/17	6/10/12/18
CarsOnBridge	3	3/5/9/17	1/3/7/18
CircArbiter	4	8/8/8/8/8	7/9/11/10/10
Choreography	1	7/13	7/17
MobileAgent	5	5/7/7/8/8/8	3/5/5/7/7/7
PressCtrller	7	5/13/17/17/21/21/21/29	2/6/8/8/10/10/11/15
ResponseCoP	3	5/14/17/17	3/4/5/6
SSFPilot	3	14/20/23/41	5/7/8/14
TrainCtrller	8	8/10/15/20/20/27/38/43/43	5/6/9/14/14/15/27/33/35

Benchmark: We provide a short description of the Event-B models in Table 3.2 including pointers where they can be retrieved:

1. *A2A* - `http://deploy-eprints.ecs.soton.ac.uk/129/` - Business domain: A model of the Order and Supply Chain A2A Communication using a pattern approach from ETH Zurich.

2. *BepiColombo* - `http://deploy-eprints.ecs.soton.ac.uk/72/` - Aerospace domain: A model of two communication modules in the embedded software on a space craft (BepiColombo mission). The model was constructed by researchers in Southampton [130] based on the feedback from SSF.

3. *CarsOnBridge* - `http://deploy-eprints.ecs.soton.ac.uk/112/` - Pedagogical example: Described in Section 3.1.3 and [94, Chapter 2].

4. *CircArbiter* - `http://deploy-eprints.ecs.soton.ac.uk/117/` - Pedagogical example: Event-B model of the synchronous electronic circuits, see also [94, Ch. 8].

Table 3.3: Results for "From scratch" and "Reuse" strategies (number of states, number of test cases and number of iterations)

			"From scratch" (i.e. no reuse)			"Reuse"		
Subject	M	ℓ	# of states	# of tests	# of iter.	# of states	# of tests	# of iter.
A2A	M2	11	5	79	5	5	79	2
	M3	11	8	285	8	8	285	5
	M4	11	8	445	8	8	445	2
	M6	11	14	1518	14	14	1518	7
BepiColombo	M2	9	47	4798	47	51	5702	11
	M3	12	126	24202	126	131	26896	65
	M2	11	48	5098	48	53	6249	13
	M3	15	131	26958	131	145	34115	75
CarsOnBridge	M1	13	5	79	5	5	79	4
	M2	13	9	429	9	9	429	4
	M3	13	38	7407	38	54	11402	39
CircArbiter	M4	12	5	141	5	7	293	2
Choreography	M0	13	5	120	5	5	120	5
	M1	13	10	957	10	10	957	5
MobileAgent	M2	12	13	445	13	17	777	2
	M3	12	25	1162	25	33	1964	18
	M4	12	25	1162	25	33	1964	2
PressCtrller	M0	8	5	81	5	5	81	5
	M1	8	43	4857	43	46	5628	41
	M2	8	127	22017	127	136	26385	92
ResponseCoP	M0	3	4	41	4	4	41	4
	M1	8	37	3230	37	43	5091	38
	M0	4	5	51	5	5	51	5
	M1	9	49	5767	49	65	9539	58
SSFPilot	M0	10	54	7541	54	54	7541	54
	M1	10	96	20430	96	96	20430	95
	M1	11	107	25370	107	107	25370	107
TrainCtrller	M5	13	12	3117	12	12	3117	2
	M6	13	12	4437	12	12	4437	2
	M7	13	18	9054	18	18	9054	15
	M8	13	20	11203	20	20	11203	17

5. *Choreography* - see [80] - Business domain: A model of service choreography for enterprise component communication.

6. *MobileAgent* - `http://deploy-eprints.ecs.soton.ac.uk/120/` - Distributed systems domain: A model for distributed computing communication: a routing algorithm for sending messages to a mobile phone, see also [94, Chapter 12].

7. *PressCtrller* - `http://deploy-eprints.ecs.soton.ac.uk/113/` - Embedded control domain: A model of a mechanical press controller adapted from a real system at INRST (Institut National de la Recherche

Table 3.4: Comparison of execution times for the original and the proposed heuristic algorithm (below, "time": execution time of the original learning algorithm, "time_h": execution time of the heuristic algorithm in Section 3.5.3, "time_tr": execution time of the transformation algorithm in Fig. 3.7)

		"From scratch" (i.e. no reuse)			"Reuse"			
Subject	M/ℓ	time	time_h	Red.%	time	time_h	Red.%	time_tr
A2A	M3/11	0.37	0.39	-4.59	0.36	0.35	2.75	0.17
	M4/11	0.80	0.93	-16.35	0.90	0.91	-1.44	0.25
	M6/11	5.92	6.36	-7.45	3.51	3.63	-3.27	3.15
BepiColombo	M2/10	29.85	25.15	15.75	38.22	32.31	15.48	2.84
	M3/12	235.87	179.92	23.72	228.85	177.64	22.38	24.29
	M2/11	30.42	25.97	14.65	40.32	35.04	13.10	2.81
	M3/15	287.40	254.56	11.43	359.01	346.03	3.62	26.59
CarsOnBridge	M1/13	0.06	0.06	0.00	0.08	0.06	30.00	0.10
	M2/13	0.90	0.87	2.90	0.81	0.74	8.74	0.38
	M3/13	34.19	33.43	2.23	48.06	46.79	2.64	5.91
CircArbiter	M3/12	0.33	0.32	2.71	0.81	0.81	0.00	0.26
	M4/12	0.39	0.39	0.00	1.10	0.96	13.00	0.34
Choreography	M0/13	0.41	0.28	32.44	0.36	0.26	28.09	0.00
	M1/13	7.08	7.16	-1.10	6.06	6.14	-1.30	2.21
MobileAgent	M2/12	1.25	1.44	-15.37	1.74	1.77	-1.61	0.80
	M3/12	7.25	7.13	1.68	11.05	11.28	-2.12	1.70
PressCtrller	M0/8	0.03	0.03	0.00	0.03	0.04	-9.38	0.00
	M1/8	5.32	5.17	2.84	5.85	5.68	2.82	0.61
	M2/8	41.71	38.16	8.52	45.66	41.63	8.84	4.03
ResponseCoP	M0/3	0.02	0.02	0.00	0.05	0.02	47.83	0.00
	M1/8	7.57	7.63	-0.79	16.75	12.69	24.24	2.28
	M0/4	0.04	0.04	0.00	0.05	0.04	8.89	0.00
	M1/9	15.93	15.64	1.81	25.10	27.49	-9.51	4.02
SSFPilot	M0/10	186.71	116.87	37.40	186.71	112.61	39.69	0.00
	M1/10	5463.69	2371.34	56.60	5463.69	2596.46	52.48	0.71
	M1/11	8435.40	3652.08	56.71	9505.11	3945.82	58.49	0.58
TrainCtrller	M5/13	31.57	33.83	-7.17	29.73	31.54	-6.07	1.38
	M6/13	99.35	110.46	-11.18	100.37	104.75	-4.37	6.76
	M7/13	394.86	451.11	-14.24	389.24	440.22	-13.10	6.78
	M8/13	531.50	536.62	-0.96	530.00	526.55	0.65	6.78

sur la Sécurité du Travail), see also [94, Chapter 3].

8. *ResponseCoP* - `http://deploy-eprints.ecs.soton.ac.uk/301/` - Information flow domain: A model for analysis of information flow policies for Dynamic Virtual Organisations (DVOs), commonly referred to as the Bronze/Silver/Gold structure that frequently arises in multi-agency response to emergencies.

9. *SSFPilot* - `http://deploy-eprints.ecs.soton.ac.uk/58/` - Aerospace domain: A model of a pilot for a complex on-board satellite mode-rich system: Attitude and Orbit Control System (AOCS).

10. *TrainCtrller* - `http://deploy-eprints.ecs.soton.ac.uk/316/` - Automotive Domain: The model specifies a controller that detects the driving mode wished by the train driver. A large number of requirements are taken into account, therefore a large number of variables and events are needed.

3.1.6 Related work

The correspondence between conformance testing and automata learning is discussed, from a theoretical point of view, by Berg et al. [85], which shows how results from one area can be transferred to the other. Such a correspondence is also exploited in our method; in our case, however, the correspondence is between conformance testing for bounded sequences and cover automata learning. Furthermore, here we propose an adaptive learning and test generation approach, in which the results obtained for a previous (incomplete or inaccurate) model are reused for the current system.

An adaptive approach to model development is proposed by Groce et al. [86], but the emphasis here is on model checking, rather than test generation, as in our approach. Furthermore, a DFA (not a DFCA) model of the system is built. Our approach can gradually build an appropriate model by suitably setting the value of the upper bound ℓ or through the Event-B refinement mechanism.

The use of automata learning techniques for test generation is also discussed by Hagerer et al. [87] and Hungar et al. [88]. As above, both papers refer to the case of unbounded sequences. Furthermore, in both papers the focus is on language learning and test generation is only mentioned as an addendum: once the model is constructed, it can be used as basis for automatic test generation. Hungar et al. present a technique for optimizing complex system learning. Hagerer et al. present a technique, called regular extrapolation, for model generation from knowledge accumulated from different sources and expert knowledge. The final model is only an approximation of the real system and so the test cases derived from it may not attain the desired level of coverage. Interestingly, testing is also used to validate the obtained model and the authors state that “most errors show up in short sequences”. This provides further support for our approach, which considers bounded sequences.

Bounded model checking (based on SAT methods) [77] is also gaining popularity in the formal verification community, the general consensus being that it works particularly well on large designs where bugs need to be searched at shallow to medium depths.

Dupont et al. [89] and Walkinshaw et al. [90] use automata inference techniques to construct behavior models of software systems. However, there are a number of key differences from our approach. Firstly, the learning techniques used are essentially passive inference methods, in which a set of training data is supplied to the algorithm for model construction. In order to construct an accurate model, training sequences which satisfy an appropriate level of coverage must be supplied. Therefore, these approaches rely on the existence of good test sets, rather than assist in producing such sets. Secondly, the behavior of all (unbounded) sequences is considered. While the model produced by this approach may be exact, it may be too complex and so the whole process may be too expensive to have a practical value. By appropriately setting the upper bound ℓ, our approach has potential to strike the right balance between accuracy and costs. Naturally, at limit (i.e. for ℓ sufficiently large), our approach will also produce an exact model of the system.

Grieskamp et al. [91] construct finite automata that under-approximate the global state space of an ASM. They use an algorithm that combine several concrete states into abstract ones using logical formulas to distinguish the abstract states. The obtained finite automaton is used for test generation. Using a similar idea and extra information on logical dependences between Event-B guards, [92] constructs an abstract over-approximation of the control flow graph of an Event-B model. Compared to our approach based on incremental learning, both these last approaches use a different approach relying on state merging and logical formulas for distinguishability.

Finally, to the best our knowledge, this is the first attempt to use grammatical inference techniques for Event-B models.

Technical appendices

Let $A = \{a_1, \ldots, a_n\}$ be an ordered set, $n > 0$. Then the quasi-lexicographical order on A^*, denoted $<$, is defined by: $x < y$ if $\|x\| < \|y\|$ or $\|x\| = \|y\|$ and $x = za_iv$, $y = za_ju$, for some $z, u, v \in A^*$ and $1 \leq i < j \leq n$. $x \leq y$ is used to denote that $x < y$ or $x = y$.

Let $U \subseteq A^*$ be a finite set and ℓ an integer that is greater than or equal to the length of the longest sequence(s) in U. Let $S \subseteq A^*$ and $W \subseteq A^*$ be the current sets processed by LearnDFCA.

For $s \in S \cup SA$, we define $r(s)$ to be the minimum sequence $t \in S$ such that $s \sim t$, where the minimum is taken according to the quasi-lexicographical order on A^*. In particular, $r(\epsilon) = \epsilon$. Then we define $S_{min} = \{r(s) | s \in S\}$.

For every two dissimilar sequences $s_1, s_2 \in S$, we denote by $d(s_1, s_2)$ the minimum element (according to the quasi-lexicographical order) of W that distinguishes between s_1 and s_2. Then we define W_{min} as the set of all such sequences, i.e. $W_{min} = \{d(s_1, s_2) \mid s_1, s_2 \in S, s_1 \not\sim_\ell s_2\}$.

Then the following result holds.

Theorem 3.1.1 *Suppose that $M(S, W, T)$, the automaton returned by LearnDFCA, is a minimal DFCA of U w.r.t. ℓ. Then the execution of LearnDFCA for inputs $S_0 = S_{min}$ and $W_0 = W_{min}$ will pass both the consistency and closedness checks and the returned DFCA $M(S_{min}, W_{min}, T_{min})$ is isomorphic to $M(S, W, T)$.*

Proof We prove by contradiction that the procedure passes the consistency check. Otherwise, there exist $w \in W_{min}$, $s_1, s_2 \in S$ with $\|s_1\|, \|s_2\| \leq \ell - \|w\| - 1$ and $a \in A$ such that $s_1 \sim_k s_2$, where $k = max\{\|s_1\|, \|s_2\|\} + \|w\| + 1$, and $T(s_1aw) \neq T(s_2aw)$. Thus aw distinguishes between s_1 and s_2, but s_1 and s_2 cannot be distinguished by any element in W_{min} of length $\|w\| + 1$. This contradicts the fact that W_{min} contains $d(s_1, s_2)$.

Similarly, if the procedure fails the closedness check, then there exist $s \in S_{min}$, $a \in A$ such that $sa \not\sim t \; \forall t \in S_{min}$ with $\|t\| \leq \|sa\|$. On the other hand $r(sa) \in S_{min}$ and $\|r(sa)\| \leq \|sa\|$. This provides a contradiction, as required.

Since the state set of $M(S, W, T)$ is S_{min}, the fact that $M(S_{min}, W_{min}, T_{min})$ and $M(S, W, T)$ are isomorphic follows directly from the definition of the DFCA returned by LearnDFCA [106].

Moreover, let A be a finite alphabet, $U \subseteq A^*$ a finite set and ℓ an integer that is greater than or equal to the length of the longest sequence(s) in U; let I be a minimal DFCA of U w.r.t. ℓ. Then the following two results hold.

Theorem 3.1.2 *If $S_0 \subseteq A^*$ is a proper state cover of I, then LearnDFCA returns a minimal DFCA of U w.r.t. ℓ for inputs S_0 and $W_0 = \{\epsilon\}$.*

Proof Let $M(S, W, T)$ be the DFCA returned by LearnDFCA. First we prove that W is a strong characterization set of I. We provide a proof by contradiction. If we assume otherwise, then there exist $w = a_1 \ldots a_i \notin W$ with $a_1, \ldots, a_i \in A$, and q_1, q_2 states of I such that q_1 and q_2 are distinguishable by w but indistinguishable by $W \cap A[i]$. Since $\epsilon \in W$, $i \geq 1$. Let j, $1 \leq j \leq i$, be the largest integer for which $a_j \ldots a_i \notin W$. Then we choose one such w for which j has the minimum possible value and q_1 and q_2 as above. Let q'_1 and q'_2 be the states reached by $a_1 \dots a_{j-1}$ from q_1 and q_2 respectively. Then q'_1 and q'_2 are distinguishable by $a_j \dots a_i$. Furthermore, q'_1 and q'_2 are indistinguishable by $W \cap A[\|i - j + 1\|]$. (Assume there exists $z = b_j \dots b_{i'} \in W$ with $b_1, \dots, b_{i'} \in A$, $i' \leq i$, which distinguishes between q_1 and q_2. Then q_1 and q_2 are distinguishable by $a_1 \dots a_{j-1} b_j \dots b_{i'}$ but indistinguishable by $W \cap A[i']$. This contradicts the minimality of j.) Let $s_1 \in S$ and $s_2 \in S$ be sequences of minimum length which reach q_1 and q_2, respectively (since S is a proper state cover, such sequences exist). Let $w' = a_{j+1} \dots a_k$, $s'_1 = s_1 a_1 \dots a_{j-1}$, $s'_2 = s_2 a_1 \dots a_{j-1}$ and $k = max\{\|s_1\|, \|s_2\|\} + i + 1$. Then $s'_1 \sim_k s'_2$ and $T(s'_1 a_j w') \neq T(s'_2 a_j w')$. This provides a contradiction as the final observation table must be consistent.

Since I and $M(S, W, T)$ have the same number of states (equal to the number of elements of S_0), the result follows from the W-method for bounded sequences.

Theorem 3.1.3 *If $W_0 \subseteq A^*$ is a strong characterization set of I, then LearnDFCA returns a minimal DFCA of U w.r.t. ℓ for inputs $S_0 = \{\epsilon\}$ and W_0.*

Proof We prove by contradiction that S is a proper state cover of I. If we assume otherwise, then there exist $s = a_1 \dots a_i \notin S$ with $a_1, \dots, a_i \in A$, $i \geq 1$, and q a state of I such that q is reached by s but cannot be reached by any sequence in $S \cap A[i]$. Let j, $1 \leq j \leq i$, be the smallest integer for which $a_1 \dots a_j \notin S$. Then we choose one such s for which j has the minimum possible value and q as above. Let q' be the state reached by $a_1 \dots a_j$ from the initial state of I. Then q' cannot be reached by any sequence in $S \cap A[j]$. (If there exists $z = b_1 \dots b_{j'} \in S$ with $b_1 \dots b_{j'} \in A$, $j' \leq j$, which reaches q' then q is reached by $b_1 \dots b_{j'} a_{j+1} \dots a_i$ but unreachable by any sequence in $S \cap A[i]$. This contradicts the minimality of j.) Let $s' = a_1 \dots a_{j-1}$. Since W is a strong characterization set of I, $s' a_j \not\sim t$ $\forall t \in S$ with $\|t\| \leq \|s' a_j\|$. This provides a contradiction. As above, the final result follows from the W-method for bounded sequences.

3.1.7 Conclusions

In this section, we presented a novel approach of using model learning for testing purposes and its application to the Event-B method. This is based on sound theoretical automata theory foundations and has an incremental and interactive nature that makes it fit the testing practice requirements. The prototype implementation showed that the method works well for realistic models, of medium or even fairly large size. As future plans, we want to further investigate the scalability of our approach and implementation on even larger models. Moreover, we plan to implement the language equivalence query as a means for interactively providing a counterexample (cf. end of Subsection 3.1.3), which is done manually in the current implementation. We will also implement different optimizations on our prototype, especially on the membership queries which constitute the most expensive part of the procedure. For instance, we can compute batches of the membership queries in parallel (on a multi-core/multi-processor architecture) or we can evaluate more sophisticated reductions, e.g. partial-order reductions by exploiting the independence of different events. In the next section, we show how we extend the model learning and test generation to decomposed Event-B models; this is important for industry, where the complexity of the large specifications is tackled not only by successive refinements but also by model decompositions.

3.2 Model-learning and MBT for Event-B decomposition

In the previous section, we proposed an iterative approach for test generation and state model inference based on a variant of Angluin's learning algorithm, which integrates well with the notion of Event-B refinement. In this section, we extend the method to work also with the mechanisms of Event-B decomposition. Two types of decomposition, i.e. shared-events and shared-variables, are considered and the generation of a global test suite from the local ones is proposed at the end of the section. The implementation of the method is evaluated on publicly available Event-B decomposed models.

3.2.1 Introduction

The main modeling approach in Event-B relies on the notion of refinement, i.e., the modeler starts with an abstract model which is iteratively enriched and concretized by capturing more and more features of the system to be specified. Each refinement step is accompanied by formal proofs for properties of interest for the system. As the complexity of the model increases, so does the difficulty the proof obligations and verification tasks. One powerful method to address this situation is to decompose a larger model into smaller sub-models which can be further refined and analyzed independently [95, 96]. There are two main types of decomposition: shared events style [97, 98] and shared variables style [99, 100]. In the former, the communication and consistency between sub-models is realized via shared events, while in the latter this is done via shared variables.

In this section, we extend the method of the previous section that integrates not only the Event-B refinement mechanism, but also the different Event-B decomposition styles. More precisely, for decomposition, we investigate the generation of CAs (cover automata [75]) for the sub-models by reusing information via projections from the global model. Also vice-versa, for the recomposition operation we can reuse the information from the CAs of the sub-models for the construction of a CA for the global model. Conformance test suites are also generated alongside. Finally, an integrated approach involving both refinements and (de)compositions in an Event-B development chain is proposed.

The section is structured as follows. The next subsection presents prerequisites from formal languages and automata theory. Subsection 3.2.3 shortly recalls the previous work on automata learning for Event-B and Subsection 3.2.4 introduces the extension of this work to Event-B decomposition and recomposition operators. Subsection 3.2.5 provides experiments on publicly available Event-B models, while Subsection 3.2.6 concludes the section.

3.2.2 Preliminaries

In this subsection we provide theoretical prerequisites product automata, together with their accepted languages.

Product automata and projections - general concepts. We now provide a couple of definitions and results for product automata and languages. This is a prerequisite for the setting of decomposed Event-B models that we present later on. To simplify the presentation, we only consider the case the two automata, but the definitions and the results hold also for more than two automata.

We start by describing formally the product of two automata synchronizing on their common input symbols. First of all, since the two automata have different input alphabets A_1 and A_2, their transition function is extended to the whole set of symbols $A = A_1 \cup A_2$ using the following definition. Given DFA $M = (B, Q, q_0, F, h)$ and $B \subset A$ we define the DFA $Ext_A(M) = (A, Q, q_0, F, h')$ by: for every $q \in Q$ and $a \in A$, $h'(q, a) = h(q, a)$ if $a \in B$ and $h'(q, a) = q$ if $a \in A \setminus B$.

When the two automata operate on the same input alphabet, their product can be described in a traditional fashion, as follows:

Definition 3.2.1 *Let $M_1 = (A, Q_1, q_{01}, F_1, h_1)$ and $M_2 = (A, Q_2, q_{02}, F_2, h_2)$ be two DFAs. Then we define the DFA $M_1 \times M_2 = (A, Q, q_0, F, h)$ by: $Q = Q_1 \times Q_2$, $q_0 = (q_{01}, q_{02})$, $F = F_1 \times F_2$ and for every $q_1 \in Q_1$, $q_2 \in Q_2$, $a \in A$, $h((q_1, q_2), a) = (h_1(q_1, a), h_2(q_2, a))$.*

Thus, for two DFAs M_1 and M_2 over alphabets A_1 and A_2, we denote by $M_1 \parallel M_2 := Ext_A(M_1) \times Ext_A(M_2)$ the *product automaton* over alphabet $A = A_1 \cup A_2$ capturing the synchronization on common symbols of M_1 and M_2. This is similar to the standard synchronization of labeled transition systems used in the literature (see e.g. [107]).

The languages accepted by product automata are characterized by the so-called *product languages*. For their definition, we first need the notion of *projection*. Given a sequence $s \in A^*$ and $A_1 \subset A$, the projection of s on A_1, denoted by $\mathit{proj}_{A_1}(s)$, is the sequence obtained from s by removing all symbols not in A_1. For a language $L \subseteq A^*$, $\mathit{proj}_{A_1}(L) = \{\mathit{proj}_{A_1}(s) \mid s \in L\}$. Now, we can define the notion of *product language*:

Definition 3.2.2 *Let A_1 and A_2 be two alphabets, not necessarily disjoint, and $A := A_1 \cup A_2$. Then, a language $L \subseteq A^*$ is called a* product language *(over A_1 and A_2) if and only if there exist two languages $L_1 \subseteq A_1^*$ and $L_2 \subseteq A_2^*$ such that*

$$L = \{w \in A^* \mid \mathit{proj}_{A_1}(w) \in L_1 \text{ and } \mathit{proj}_{A_2}(w) \in L_2\}.$$

Moreover, there exist also a useful result (see e.g. [108]) proving that a product language is always the product of its projections, i.e. languages L_1 and L_2 in the previous definition can be replaced by $\mathit{proj}_{A_1}(L)$ and $\mathit{proj}_{A_2}(L)$, respectively. Finally, the expected result relating the languages of product automata with product languages says that:

Proposition 3.2.3 *[108] The class of regular product languages coincides with the class of languages accepted by products of DFAs.*

Corollary 3.2.4 *For a finite alphabet $A := A_1 \cup A_2$, let $L \subseteq A^*$ be a regular product language, and M_1 and M_2 be two DFAs for $\mathit{proj}_{A_1}(L)$ and $\mathit{proj}_{A_2}(L)$, respectively. Then, $L = L_{M_1 \| M_2}$.*

Since any finite language is also a regular language, Corollary 3.2.4 holds also when L is a finite product language. Therefore, we can easily derive:

Corollary 3.2.5 *For a finite alphabet $A := A_1 \cup A_2$, let $U \subseteq A^*$ be a finite product language and ℓ a positive bound (larger than the size of any word in U). If M_1 and M_2 are two DFCAs w.r.t. ℓ for $\mathit{proj}_{A_1}(U)$ and $\mathit{proj}_{A_2}(U)$, then $M_1 \parallel M_2$ is a DFCA w.r.t. ℓ for U.*

3.2.3 Cover automata based learning and test generation

In this subsection we present the main elements of the approach proposed in [104], that can incrementally construct a series of finite state approximations and corresponding test suites for a series of Event-B refined models. Before that, we need to provide the basic elements of Event-B.

A short introduction to Event-B.

An event is an element consisting of a set of *local parameters*, a *guard* and an *action* code. An event *evt* has the following general form:

$$\mathit{evt} \;\widehat{=}\; \textbf{any } t \textbf{ where } G(t, v) \textbf{ then } S(v, t) \textbf{ end}. \tag{3.1}$$

Above, t is a set of local parameters, v is a set of global variables appearing in the event, G is a predicate over t and v, called the guard, and $S(v, t)$ represents a substitution. If the guard of an event is false, the event cannot occur and is called disabled. The substitution S modifies the values of the global variables in the set v. It can use the old values from v and the parameters from t. For example, an event that adds a number i smaller than 9 to a global variable n, in case n is greater than 15, is modeled as:

$$\mathit{increment} \;\widehat{=}\; \textbf{any } i \textbf{ where } i \in \mathbb{N} \;\wedge\; i < 9 \;\wedge\; n > 15 \textbf{ then } n := n + i \textbf{ end}.$$

Given an Event-B model, *a test case* can be defined as a sequence of events. This can be either positive, if it corresponds to a feasible (i.e. executable) path through the Event-B model, or negative, otherwise. The feasibility of a test case implies the existence of appropriate *test data* for the events, i.e. an appropriate initialization of the global variables and suitable values for the local parameters,such that all the guards of the events in the sequence are satisfied. Furthermore, a *test suite* is by definition a collection of test cases.

Given an Event-B model Z having its set of events denoted by E, we can define the language of Z to be the set of feasible sequences over E, i.e.

$$L(Z) := \{w \in E^* \mid w \text{ is feasible in } Z\}.$$

Note that $L(Z)$ is not regular in general, since one can easily simulate a two-counter machine in Event-B, so the formalism is Turing-complete [81]. However, we can naturally obtain a regular subset by considering only a finite subset of $L(Z)$, namely the sequences of length up to a bound ℓ, i.e. $L(Z, \ell) := L(Z) \cap E[\ell]$.

Finally, the *refinement in Event-B* is a mechanism of constructing a series of more abstract models before reaching a very detailed one. For instance, in a refinement step, new variables and new events can be introduced and the existing events can be made more concrete with the assumption (that must be formally proved) that the concrete guard is not weaker than the abstract one (i.e. the concrete guard logically implies the abstract one) [94].

Incremental model learning based on cover automata. In [104] we present an automata learning and test generation procedure for Event-B: given an Event-B model Z and a positive bound ℓ, we produce a DFCA M for $U := L(Z, \ell)$ and an associated test suite. The procedure can be iteratively used for a series of model refinements.

The core of the procedure is based on a modification of Angluin's learning algorithm [106] that is specialized to finite languages, and that is more efficient than the original Angluin's algorithm, called L^*, for regular languages [74].

In a similar but not trivial way, in [106] we extend Angluin's work by proposing an algorithm, called L^ℓ, for learning a DFCA. Given an unknown finite set $U \subseteq A^*$ and a known integer ℓ that is greater than or equal to the length of the longest sequence(s) in U, the L^ℓ algorithm will construct a minimal DFCA of U w.r.t. ℓ. Analogously to L^*, the L^ℓ algorithm uses membership and language equivalence queries to find the automaton in polynomial time.

The L^ℓ algorithm constructs two sets: S, a non-empty, prefix-closed set of sequences and W, a non-empty, suffix-closed set of sequences. Additionally, S will not contain sequences longer than ℓ and W will not contain sequences longer than $\ell - 1$, i.e. $S \subseteq A[\ell]$ and $W \subseteq A[\ell - 1]$. The algorithm keeps an *observation table*, which is a mapping T from a set of finite sequences to $\{0, 1, -1\}$. The sequences in the table are formed by concatenating each sequence of length at most ℓ from the set $S \cup SA$ with each sequence from the set W. Thus, the table can be represented by a two-dimensional array with rows labeled by elements of $(S \cup SA) \cap A[\ell]$ and columns labeled by elements of W. The function $T : ((S \cup SA) \cap A[\ell])W \longrightarrow \{0, 1, -1\}$ is defined by $T(u) = 1$ if $u \in U$, $T(u) = 0$ if $u \in A[\ell] \setminus U$ and $T(u) = -1$ if $u \notin A[\ell]$. The values 0 and 1, respectively, are used to indicate whether a sequence is contained in U or not. However, only sequences of length less than or equal to ℓ are of interest. For the others, an extra value, -1, is used. Similar to the L^* algorithm, two properties of the observation table are defined: *consistency* and *closedness*.

The algorithm starts with $S = W = \{\epsilon\}$. It periodically checks the consistency and closedness properties and extends the table accordingly using membership queries. When both conditions are met, the DFA $M(S, W, T)$ corresponding to the table is constructed and it is checked whether the language L accepted by $M(S, W, T)$ satisfies $L \cap A[\ell] = U$. If this language query fails, a counterexample t is produced, the table is expanded to include t and all its prefixes and the consistency and closedness checks are performed once more. Eventually, the language query will succeed and the algorithm will return a minimal DFCA of U w.r.t. ℓ.

The iterative procedure of the algorithm for Event-B is shortly presented below. The technical details can be found in [104]. The main idea is that we evolve the observation table based on previous versions of it, by reusing existing information whenever possible. In particular, for the Event-B refinement, the observation tables of the refined model is not generated from scratch, but from the table of the abstract model that is refined, so unlike the original L^ℓ algorithm, the procedure does not start with empty S, W and T, but with some initial values S_0, W_0 and T_0, which reflect the current knowledge about the DFCA model. An important observation is that, for efficiency reasons, in the recalculation of the observation table only a part of the previous information is sufficient, viz. $S_{min} \subseteq S$ and $W_{min} \subseteq W$, which satisfy certain properties: they are a proper state cover and strong characterization set, respectively (see [104] for definitions).

For the first execution of the procedure, the initial sets S_0 and W_0 are based on an initial estimation of the states of the model. In the worst case (when no initial estimation is available), we take $S_0 = \{\epsilon\}$, $W_0 = \{\epsilon\} \cup E$, where E is the set of events. Note that the alphabet A from L^ℓ above is now the set E. When the procedure has been applied at least once, previous information can be reused. If model is not totally accurate and needs to be improved, we can distinguish the different reasons for that:

- **Case 1:** the Event-B model has been modified or augmented due to changes in the requirements.

- **Case 2:** the Event-B model has not been changed but the associated DFCA is deemed to be insufficient for testing purposes. In this case, the upper bound ℓ is increased according to the existing testing needs and the procedure is executed once more for the new value of ℓ.

– **Case 3:** the existing Event-B model has been refined and extra detail has been added (using the Event-B refinement). In this case, information from the abstract model can be reused in the computation of the refined model.

A test suite TS can be derived from the observation table as follows:

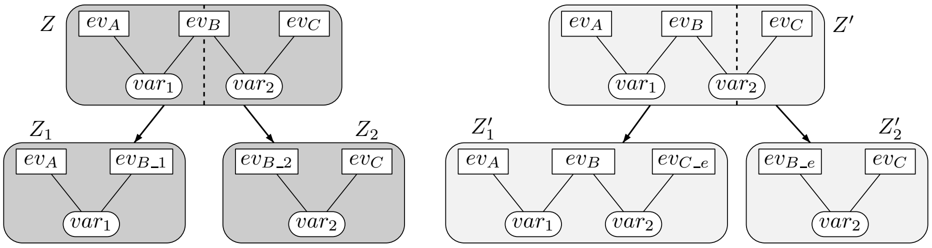
$$TS := \{t \in E^* \mid t \in ((S \cup SE) \cap E[\ell])W \text{ such that } T(t) = 1\}. \tag{3.2}$$

Note that we only take positive test cases into account in TS. However, we could also use the existing information about infeasible sequences, i.e. T(t)=0, to generate negative tests, if such a testing requirement exists. Moreover, in (3.2) we usually take S and W to be the sets S_{min} and W_{min} mentioned above. Furthermore, the test cases from TS are provided with the test data that prove their feasibility. The test data is obtained during the construction of the observation table T, because the membership queries, i.e. feasibility checks, are implemented using a dedicated set-based constraint solver for Event-B, which also returns the values of variables and local parameters for a given feasible sequence. As discussed in [104], TS will constitute a conformance test suite for the Event-B model modulo the bound ℓ (the ℓ-bounded behavior of the model). Such a test suite is more powerful than test suite based on simple state or transition coverage criteria since it covers all states and all transitions of the equivalent automaton and also checks each state and the initial and destination states of each transition. Conformance testing is especially relevant in the embedded systems domain.

3.2.4 Model learning for Event-B decomposition

Event-B decomposition styles

There are two main decomposition styles in Event-B: shared-events [97, 98] and shared-variables [99, 100]. Other variants like *atomicity decomposition* [97, 130] or *modularization* [109, 96] also exist, but we do not address them now for the following reasons. Since the atomicity decomposition is in fact a special case of refinement, our method in [104] works for it out-of-the-box. On the other hand, modularization defines a different approach to decomposition that reuses a sub-model in several other models using interface specifications, so we leave its investigation to the future (moreover, there is some yet to be solved integration issues between the modularization plug-in and the Event-B constraint solver that we use).



(a) shared event decomposition

(b) shared variable decomposition

Figure 3.8: The shared event vs. shared variable decomposition styles

Shared events decomposition. In the case of shared events decomposition, an Event-B model is decomposed into several sub-models such that all its events and variables are distributed over the local models. As the name suggests, the local sets of events may have common events (shared events). However, the local sets of variables are disjoint, i.e. the partition of the variables will determine the structure of the decomposition. The left hand side of Fig. 3.8 presents a minimalistic example of shared events decomposition. At the top, we have a global model Z with three events $\{ev_A, ev_B, ev_C\}$ and two global variables $\{var_1, var_2\}$. The lines between the events and variables suggests the dependencies between them, e.g $ev_A - var_1$ means that var_1 appears in the guard or/and action of ev_A. Assume that the modeler chose to distribute the variables over two sub-models: the first one, denoted Z_1, takes over var_1, and the second, Z_2, takes over var_2. Then, the events are distributed to Z_1 and Z_2 according to the distribution of the variables, so Z_1 has ev_A and ev_B as events (because they depend on var_1) and similarly, Z_2 has ev_B and ev_C as events. In this case, ev_B is a shared event for Z_1 and Z_2.

However, there is a technical issue to be solved for ev_B; the fact that ev_B depends on both var_1 and var_2, while the local models contain only one of the variables. This means that the local events corresponding to ev_B, denoted in Fig. 3.8 by ev_{B_1} and ev_{B_2}, will only be restricted versions of ev_B that only depend on var_1 and var_2, respectively. So, for the decomposition to be possible, ev_B should have such a form that "separates" the use of var_1 and var_2 in its guards and actions. This is a task for the modeler that should design the Event-B specification in this way as a preparation step for decomposition (refinement may be use in previous modeling steps to achieve this goal). Below, we present ev_B, ev_{B_1}, and ev_{B_2} using the general form of an event in (3.1):

$$\begin{array}{lcll}
ev_B & \widehat{=} & \textbf{any } t, t_1, t_2 & \textbf{where } G_1(t, t_1, var_1) \wedge G_1(t, t_2, var_2) \\
 & & & \textbf{then } S_1(var_1, t, t_1); S_2(var_2, t, t_2) \textbf{ end.} \\
ev_{B\_1} & \widehat{=} & \textbf{any } t, t_1 & \textbf{where } G_1(t, t_1, var_1) \textbf{ then } S_1(var_1, t, t_1) \textbf{ end.} \\
ev_{B\_2} & \widehat{=} & \textbf{any } t, t_2 & \textbf{where } G_2(t, t_2, var_2) \textbf{ then } S_2(var_2, t, t_2) \textbf{ end.}
\end{array}$$

Above, we see that ev_B has a set of local parameters t, t_1, t_2, a guard that is the conjunction of two guards using var_1 and var_2 separately, and also an action that can be split into two actions that do not mix the two global variables. The local events will then only use the parts of the guards and actions that refer to their corresponding global variable. Without going into details, it is also important to observe the existence of the common local parameter t, which can be used for passing data between ev_{B_1} and ev_{B_2}. This makes the shared event decomposition suitable for specifying distributed systems communicating via message-passing [130]. Finally, we mention also the fact that the decomposition mechanism is correct in the sense of Event-B refinement [94], by proving specific proof obligations (e.g. deadlock freedom) and putting restrictions on the subsequent refinements of the shared events in the local sub-models.

The decomposition operation induces the inverse operation of *composition*, for which a dedicated Rodin plug-in exists [110]. It takes a input two models Z_1 and Z_2 that may have events with the same name and constructs a composed model Z (look at Fig. 3.8 bottom-up). Z is obtained by putting together the variables and events Z_1 and Z_2, taking care that the local shared events are merged by concatenating their guards and actions following the same scheme as for ev_{B_1}, ev_{B_2}, and ev_B above.

Shared variables decomposition. Let us also touch upon the shared variables decomposition, using the exemplification on the right of Fig. 3.8. In this case, we partition the set of events and then distribute the variables. If we partition the events of Z' into $\{ev_A, ev_B\}$ and $\{ev_C\}$, due to the variables dependences, the sub-models Z'_1 and Z'_2 have the variables $\{var_1, var_2\}$ and $\{var_2\}$, so they share variable var_2. However, since sub-models have in fact two copies of the shared variable, they need to learn the changes made to the shared variable in the other sub-models. This is implemented adding so-called *external* events. For instance, in addition to its "native" event ev_C, Z'_2 will also include an external event ev_{B_e} that is a restricted version of ev_B, that only simulates its effect on var_2. Note that the shared variables decomposition is suitable for the specification and verification of parallel programs [100].

Learning and test generation for shared events decomposition

In the rest of the section, we will present our approach only for the shared events decomposition. We can do this without loss of generality based on the observation that, for our purposes, the shared variables decomposition can be reduced to the shared event decomposition as follows. Suppose Z' is decomposed using shared variables into Z'_1 and Z'_2 and the decomposition is based on the partition of set of events E of Z' into E_1 and E_2 Assume that $E_{11} \subseteq E_1$ is the set of external events for Z_1 and $E_{21} \subseteq E_2$ the set of external events for Z_2. Then, if we duplicate the shared variables and consider each of the two Event-B components to work on its own copy (the definition of the shared variables ensures that they process the two copies identically), the shared variables decomposition can be transformed into a shared events decomposition of Z' into sub-models with set of events $E'_1 = E_1 \cup E_{21}$ and $E'_2 = E_2 \cup E_{12}$, respectively.

Before we proceed, we establish a formal relation between Event-B decomposition and the theory of product languages from Subsection 3.2.2. The proofs of the theoretical results can be found in the Appendix.

Lemma 3.2.6 *Let Z be an Event-B model, which is decomposed into Z_1 and Z_2. Then, for any w sequence of events in Z, w is feasible if and only if, $proj_1(w)$ and $proj_2(w)$ are both feasible in Z_1 and Z_2, respectively.*

Proof We assume a shared event type of decomposition, because the shared variable decomposition can be reduced to this case as mentioned at the beginning of Section 3.2.4. Moreover, suppose that set of global variables of Z, denoted by V, is partitioned into V_1 and V_2 in the Z_1 and Z_2 components.

For the direct implication, if $w = e_1 \dots e_n$ is feasible, there exists an initialization of variables and choice of the local parameters of the events such that all the guards of e_1 to e_n are true. The property of the events occurring in $proj_1(w)$ is that they only operate on the variables from V_1. Choosing the same initialization of variables in V_1 and the local parameters as in the execution of w, it is easy to see that the guards of the events of $proj_1(w)$ are also true and so $proj_1(w)$ is feasible. The argument is similar for the second projection $proj_2(w)$.

The inverse implication is also simple, although there is a subtle observation to be made at the end of the proof. Provided that $proj_1(w)$ and $proj_2(w)$ are feasible, we want to show that w is feasible. Since V_1 and V_2 form a partition of V, we can choose an initialization of the variables in V based on the initializations of V_1 and V_2. Moreover, we know that there exist values of the local variables to satisfy the guards of $proj_1(w)$ and $proj_2(w)$. With these values, it can be proved that w is also feasible. This takes into account the fact that for a shared events e, its partial versions e_1 and e_2 in the components are merged by combining the guards with the "and"-operator and concatenating the actions.

In the inverse implication, the move from the local feasible projections to the global feasible sequence involves also initialization of the local parameters. A special case is that of common local parameters for shared events (recall the presentation from Subsection 3.2.4). More precisely, we need to make sure that there are no local initializations of a common parameter in the sub-models that are inconsistent. E.g. an initialization of $t = 1$ in one component and $t = 2$ in the other component, would lead to a deadlock in the shared event. However, by design, the definition of local parameters excludes this situation as discussed in [98]. There, the authors specify certain non-blocking usage patterns for the common parameters like input-output data passing.

Finally, as expected, the above proof can be generalized from two to more components. Note also that in Lemma 3.2.6, the projection operator uses in the case of shared events, their local versions (see ev_B, ev_{B_1}, and ev_{B_2} in Fig. 3.8). □

Using Lemma 3.2.6 and Definition 3.2.2 for product languages, we can show that:

Proposition 3.2.7 *Let Z be an Event-B model, which can be decomposed into Z_1 and Z_2. Then, the language of Z, $L(Z)$, is a product language over $E := E_1 \cup E_2$, where E_1 are the events of Z_1 and E_2 are the events of Z_2.*

Proof We prove using Lemma 3.2.6 that $L(Z)$ is a product language when choosing L_1 and L_2 in the definition of product language to be $L(Z_1)$ and $L(Z_2)$, respectively. □

As an immediate corollary, the result hold also when we impose a bound ℓ, i.e. $L(Z, \ell)$ is also a product language, so Corollary 3.2.5 can be applied.

Next we now show how our learning and test generation method can be applied to the two operations of decomposition and composition.

Approach for decomposition. Let Z be an Event-B model and E the set of events of Z. We assume that Z is decomposed, using the shared events scheme, into models Z_1 and Z_2 with event sets E_1 and E_2, respectively. Given a bound ℓ, our goal is to obtain DFCAs and associated test suites for Z, Z_1, and Z_2. Although one can apply the method in Subsection 3.2.3 directly and separately on Z, Z_1, and Z_2, we would like to improve the process by reusing information.

We assume that we have a DFCA M and a test suite TS for Z. Then, the DFCA learning procedure for Z_1 will not start with $S_1 = \{\epsilon\}$, as when no previous model is available, but with the set $S_1 = \{proj_1(s) \mid s \in S_{min}\}$, where S_{min} is the proper state cover derived from the DFCA model of Z. The set W_1 is initialized with $E_1 \cup \{\epsilon\}$. Similarly for Z_2. We could also to start with a projection of the set W obtained for Z (i.e. $W_1 = \{proj_1(s) \mid s \in W_{min}\}$), but, this may not improve performance since W usually contains only singletons [104].

With this input, the learning procedure may not produce a correct DFCA M_1 for Z_1 from the beginning and more iterations may be needed. The reason is that, even though Lemma 3.2.6 ensures that a feasible path in Z is projected to a feasible path in Z_1, the projection S_1 may not be rich enough to cover all the states of M_1. This can be understood from the fact that, in general, there is no concrete relation between the sizes of a minimal DFA of a regular language $L \subseteq A^*$ and of the minimal DFA of its projection on a sub-alphabet $A' \subset A$. Thus, the size of a minimal DFA accepting the projection $\mathit{proj}_{A'}(L)$ can be smaller, equal to, or even exponentially larger than the size of the minimal DFA accepting L [111]. The same holds even when L is a finite language. Moreover, in the specific case of Event-B decomposition, the DFCAs of the sub-models may be larger not only because of the effects of the projections just mentioned, but also because there might exist more feasible paths in the projections due to the weakened guards of the shared events, with the effect that the DFCAs for the local sub-models have more states. However, our experiments showed that our choice of S_1 will speed up the learning procedure, generating richer DFCAs in less time compared to the procedure of learning an DFCA for Z_1 from scratch.

Approach for composition. The inverse operation to decomposition is that of composition [110, 98]. Given two models Z_1 and Z_2 with event sets E_1 and E_2, one can construct an Event-B model Z that synchronizes on the shared events.

There are several ways in which we can construct a global DFCA model and/or a test suite for Z from Z_1 and Z_2 or their DFCAs:

1. Construct Z and then apply the techniques of [104] to derive a DFCA and a test suite associated to Z. In this case, there is no reuse of information from Z_1 and Z_2.

2. Construct the two DFCAs M_1 and M_2 for Z_1 and Z_2 and then construct the product $M_1 \parallel M_2$, minimize it and denote it M_{min}. Then, construct a test suite TS from the minimal DFCA M_{min} using a W-method adapted to bounded testing [103]. For every test sequence s for M_{min}, the test data generation process will check if $\mathit{proj}_1(s)$ and $\mathit{proj}_2(s)$ are test sequences for M_1 and M_2, respectively. If this is the case, the test data values for $\mathit{proj}_1(s)$ and $\mathit{proj}_2(s)$ will be reused. This variant is sound due to Corollary 3.2.5.

3. Construct only a global test suite TS from the local test suites TS_1 and TS_2 by composing individually the test cases, i.e. $TS := \{t \in E^* \mid proj_1(t) \in TS_1 \text{ and } proj_2(t) \in TS_2\}$. (Optionally, apply a symmetry reduction by only keeping traces in TS that are not equivalent modulo swapping of independent events.)

4. Construct directly a DFCA for the composed model Z without applying the composition of Z_1 and Z_2, nor the product of M_1 and M_2. This is done by applying a learning algorithm for global sequences of events (of length up to ℓ) and answering the global membership queries via answering the local membership queries for the projections (this is sound because of Lemma 3.2.6).

$$\begin{tikzcd}
 & & RZ_1 \arrow[r] & RRZ_1 \\
Z \arrow[r] & RZ \arrow[ur] \arrow[dr] & & \\
 & & RZ_2 \arrow[r] & RRZ_2
\end{tikzcd}$$

Figure 3.9: A sample of decomposition flow

The first two proposals above are correct, i.e. the obtained automata are DFCAs with respect to $L(Z, \ell)$, while the last two are heuristics that in our experiments produced reasonable results, even though they are in general only approximations.

Approach for integrated process. Finally, let us sketch how the above proposals can be integrated in our incremental, refinement based, model learning and test generation strategy presented in [104].

Figure 3.9 describes a typical incremental development in Event-B involving decomposition. There, RZ, which is a refinement of Z, is decomposed into RZ_1 and RZ_2, which are further refined into RRZ_1 and RRZ_2. For this example, our approach will first construct a DFCA model for Z, which will be reused in the construction of a DFCA for RZ. RZ will constitute the basis for the construction of the DFCAs for RZ_1 and RZ_2 starting the learning procedure with the projections as previously explained. The DFCAs for RZ_1 and RZ_2 will, in turn, be reused in the construction of the final models, for RRZ_1 and RRZ_2. These latter models are used to produce a DFCA model and tests for the overall system by one of the methods proposed for the composition operator.

3.2.5 Experiments

We implemented the methods for decomposition presented in this section, extending our Rodin plug-in that previously only addressed refinement [105]. The experiments were conducted on a Windows 7 Professional 64-bit machine with an Intel Core i7 2.80GHz (8 CPUs) processor and 12 GB of RAM.

For the benchmark, we investigated all the publicly available Event-B models involving decomposition from the DEPLOY repository[10]. From the total of eight found models, we could not use two of them because they involved some advanced data types that were not yet supported by the Event-B constraint solver deployed for the membership queries. From the rest of six models, the first three use shared events and the last three use shared variables. Their dimensions are presented in Table 3.5. The first column gives their name together with a reference. The second column gives the evolution of the models by the operations of

[10] `http://deploy-eprints.ecs.soton.ac.uk`

refinement and decomposition in a similar fashion to Fig. 3.9. The '/' symbol represents a refinement step, while '{' depicts a decomposition. For instance, for BepiColombo_SE, there are three refinement steps $m_0/m_1/m_2/m_3$, followed by a decomposition of m_3 into m_4 and m_5; then, m_4 is further refined to m_6 and m_7. The third and forth columns provide the corresponding numbers of events and global variables for the models. For example, BepiColombo starts at m_0 with 6 events and 5 global variables, increases its complexity via refinement to m_3 which exhibits 17 events and 16 variables. and ends up having 23 events and 20 variables for the last refinement m_7 of one of the sub-models.

Note that the search space in BepiColombo case can be very large. E.g. the third refinement m_3 of BepiColombo has 17 events, so for $\ell := 8$ the number of possible sequences or tests of length up to 8 is 17^8 which is almost equal to $7 \cdot 10^9$. Moreover, to this complexity we have to add the computation time for test data for the generated test cases. The constraint solver performing this task need to address a search space implied by 16 global variables of type Set and 17 local parameters appearing in the events.

In our experiments, we checked the feasibility of our approach and the scalability of the implementation, by performing the steps for the integrated process at the end of the previous subsection, i.e. we incrementally construct DFCAs for the refinement and decomposition from abstract model to more concrete levels, combining the (integration) tests at the end using a method for composition. Due to space constraints, we provide the tables with experimental results in the Appendix. However, we report a successful generation of DFCAs and test suites within reasonable time (max. 6 minutes) for sufficiently high values of ℓ (up to 13 for smaller models). Moreover, the experiments confirmed that the reuse improves the quality of the generated DFCAs (i.e. more states compared to learning from scratch) and reduces the computation time in many cases.

3.2.6 Conclusions

In this section, we presented a method for automata learning and test generation that can be applied along the specification process of Event-B. We focused on the mechanism of decomposition, because this is an important way of dealing with the large models that may occur in industrial practice. Our approach makes use of the advantages of cover automata and its soundness is based on the theory of product languages. In the future, we will continue to improve the prototype e.g. by a better (UI) integration with decomposition and composition plug-ins [95, 110] and extend its use to the modularization plug-in [109]. We will also investigate the quality of the generated test suites using mutation testing techniques.

In the end, we mention a couple of related papers, even though they solve different problems in different settings. First, we are not aware of any work that

Table 3.5: The dimensions of 6 models from DEPLOY repository (development process, no. of events and no. of variables)

Subject	**Development process**	**No. events** $(m_0/m_1\ldots)$	**No. variables** $(m_0/m_1\ldots)$
BepiColombo_SE from [130]	$m_0/m_1/m_2/m_3 \begin{cases} m_4/m_6/m_7 \\ m_5 \end{cases}$	$6/11/13/17 \begin{cases} 15/19/23 \\ 10 \end{cases}$	$5/10/12/16 \begin{cases} 12/16/20 \\ 4 \end{cases}$
UpdateMaster_SE from [96]	$m_0/m_1/m_2 \begin{cases} m_3/m_5/m_7 \\ m_4/m_6/m_8 \end{cases}$	$5/6/6 \begin{cases} 4/5/5 \\ 4/6/6 \end{cases}$	$4/5/5 \begin{cases} 4/6/6 \\ 3/8/8 \end{cases}$
Monitor_SE from [96]	$m_0/m_1/m_2 \begin{cases} m_3/m_6/m_9 \\ m_4/m_7/m_{10} \\ m_5/m_8/m_{11} \end{cases}$	$7/7/7 \begin{cases} 7/5/5 \\ 4/6/6 \\ 4/6/6 \end{cases}$	$4/6/6 \begin{cases} 2/4/4 \\ 2/3/3 \\ 2/3/3 \end{cases}$
Monitor_SV from [96]	$m_0/m_1/m_2 \begin{cases} m_3/m_6/m_9 \\ m_4/m_7 \\ m_5/m_8/m_{10} \end{cases}$	$7/11/11 \begin{cases} 9/11/11 \\ 10/10 \\ 7/7/9 \end{cases}$	$4/4/4 \begin{cases} 2/5/5 \\ 3/4 \\ 3/4/6 \end{cases}$
QResponse_SV	$m_0/m_1/m_2/m_3/m_4 \begin{cases} m_5 \\ m_6 \\ m_7 \end{cases}$	$2/3/4/5/5 \begin{cases} 3 \\ 5 \\ 3 \end{cases}$	$2/3/5/7/9 \begin{cases} 4 \\ 7 \\ 4 \end{cases}$
FindP_SV from [100]	$m_0 \begin{cases} m_1/m_3/m_4/m_5 \\ m_2 \end{cases}$	$6 \begin{cases} 4/5/6/6 \\ 4 \end{cases}$	$5 \begin{cases} 3/4/5/6 \\ 3 \end{cases}$

generates test cases for Event-B decomposed models, see e.g. [112] and the references therein. An idea of using model projections combined with automata learning for black-box testing of components is presented in [113]. Our relation between learning and conformance test suite is similar to the one presented in [85]. Learning is also used for the generation of communicating automata [114, 115] and for compositional verification of system components [107].

Table 3.6: Side-by-side results for the "Reuse Information" and "No Reuse" strategies for refinement decomposition. We give information on: $|DFCA|$ – the number of states of the DFCA, $|TS|$ – the number of test cases, and the execution times in seconds. Note that the "reuse" method generates richer DFCAs for BepiColombo_SE and QResponse_SV (and same sizes for the rest) and has better running times in 60% of the lines below.

			"Reuse Information"			"No Reuse"		
Subject	Z	ℓ	$\|DFCA\|$	$\|TS\|$	time	$\|DFCA\|$	$\|TS\|$	time
BepiColombo_SE	m_0	3	5	9	0.05	5	9	0.07
	m_1	4	10	28	1.24	10	28	0.82
	m_2	4	11	36	2.07	11	36	1.67
	m_3	5	19	65	10.79	19	65	9.14
	m_4	5	105	1,966	19.16	103	1,891	22.12
	m_5	5	28	367	3.26	25	318	2.79
	m_6	6	358	7,635	267.37	351	7,294	292.92
	m_7	6	471	10,039	387.53	465	9,687	837.87
UpdateMaster_SE	m_0	12	5	5	0.06	5	5	0.06
	m_1	12	5	7	0.14	5	7	0.10
	m_2	12	5	7	0.28	5	7	0.11
	m_3	12	4	6	0.02	4	6	0.03
	m_4	12	3	4	0.01	3	4	0.01
	m_5	12	5	7	0.16	5	7	0.08
	m_6	12	5	7	0.17	5	7	0.09
	m_7	12	6	7	0.20	6	7	0.17
	m_8	12	6	7	0.19	6	7	0.19
Monitor_SE	m_0	11	4	9	0.10	4	9	0.06
	m_1	11	5	11	0.33	5	11	0.12
	m_2	11	5	11	0.02	5	11	0.15
	m_3	11	5	11	0.00	5	11	0.15
	m_4	11	4	6	0.00	4	6	0.03
	m_5	11	3	4	0.01	3	4	0.03
Monitor_SV	m_2	12	13	44	0.22	13	44	1.33
	m_3	12	13	34	0.04	13	34	0.9
	m_4	12	13	39	0.00	13	39	1.10
	m_5	12	5	7	0.00	5	7	0.12
	m_6	12	13	34	0.20	13	34	1.45
	m_7	12	13	34	0.08	13	34	1.30
	m_8	12	13	39	0.10	13	39	1.26
	m_9	12	5	7	0.02	5	7	0.12
	m_{10}	12	5	7	0.05	5	7	0.21
QResponse_SV	m_1	13	4	6	0.09	4	6	0.02
	m_2	13	6	10	0.21	6	10	0.07
	m_3	13	9	18	0.49	9	18	0.25
	m_4	13	9	18	0.26	9	18	0.34
	m_5	13	5	4	0.01	3	3	0.05
	m_6	13	11	33	0.18	7	15	0.11
	m_7	13	3	2	0.01	3	2	0.01
FindP_SV	m_0	6	5	15	0.15	5	15	0.16
	m_1	6	3	5	0.01	3	5	0.02
	m_2	6	3	5	0.01	3	5	0.04
	m_3	6	6	34	35.10	6	34	0.22
	m_4	6	6	61	117.60	6	61	45.55
	m_5	6	6	30	0.52	6	30	0.34

Table 3.7: The number of generated test suites by the heuristics at the end of Subection 3.2.4, i.e (3) the test suite TS_s obtained from the synchronization of local test suites, and (4) the test suite TS_h by the heuristic of constructing a direct DFCA by answering global membership queries via local ones. The times were reasonable: under 2 seconds for all except TS_i of BepiColombo_SE where 18 minutes were needed to compute the 3 millions test cases (these large number is obtain because TS_s is obtain from the synchronization of large local test suites: 10,039 for m_7 and 367 for m_5 – see previous table). The values of ℓ were those used also in Table 3.6.

Subject	Machines	$\|TS_s\|$	$\|TS_h\|$
BepiColombo_SE	$m_7\|\|m_5$	3,097,890	220
UpdateMaster_SE	$m_7\|\|m_8$	37	9
Monitor_SE	$m_3\|\|m_4\|\|m_5$	145	11
Monitor_SV	$m_9\|\|m_7\|\|m_{10}$	4,527	38
QResponse_SV	$m_5\|\|m_6\|\|m_7$	40	18
FindP_SV	$m_5\|\|m_2$	135	24

3.3 Meta-heuristic methods for test data generation for Event-B

3.3.1 Introduction

The ProB tool [131] has a good control of the state space, being able to explore it, visualize it and verify various properties using model-checking algorithms. Such algorithms can be used to explore the state space of Event-B models using certain coverage criteria (e.g. event coverage) and thus generating test cases along the traversal. Moreover, the input data that trigger the events provides the test data associated with the test cases. Such an approach using explicit model-checking has been applied to models from the business application area by SAP [80]. The algorithms perform well for models with data with a small finite range. However, in case of variables with a large range (e.g. integers), the known state space explosion problem creates difficulties, since the model checker explores the state space by enumerating the many possible values of the variables.

This section addresses a slightly different, but related, problem. Given a (potentially feasible) path in the Event-B model, we use meta-heuristic search algorithms (more precisely, genetic algorithms) to generate input data that trigger the execution of the path. This is a very important issue of MBT since, for models with large state spaces, paths with restrictive triggering conditions (e.g. composed conditions involving one or more = operators) are difficult to attain using the model checking approach described above. A similar problem has been addressed by recent work on search-based testing for Extended Finite State Machines (EFSMs) [139, 136, 149]. However, there are a number of issues that differentiate search-based testing on Event-B models from these EFSM approaches as described our position paper [146] like implicit state space, non-numerical types, non-determinism and hierarchical models. In this section, we start addressing some of these issues, especially the non-numerical types.

The main contributions of this section are enumerated below:

- Since the data structures used by Event-B models are predominantly set-based rather than numerical, Tracey-like [147] fitness functions for such data types are newly defined. These fitness functions are used to guide the search for the solutions in large state spaces.

- Furthermore, the encoding of non-numerical types into a chromosome is investigated. As Event-B models may use a mixture of numerical and non-numerical types, the encoding has to accommodate also such a possibility.

- The proposed search-based testing approach for Event-B is applied on a number of industry-inspired case studies. The experiments show that the approach performs better in general compared to random testing approaches.

The section is structured as follows. We start with a couple of representative Event-B examples in Subection 3.3.2. Then we present the proposed test generation framework based on search-based techniques using genetic algorithms in Subsection 3.3.3. The experiments are explained in Subsection 3.3.4 and the conclusions are drawn in Subsection 3.3.5.

3.3.2 Case studies

In Section 3.5.3 we run the experiments on a benchmark of 5 Event-B models. The models are not industrial ones, but are inspired by industrial examples. We have been in contact with partners in the DEPLOY project that are interested in test generation from Event-B models, especially SAP, which is an industrial partner from the business software area. We have discussed a couple of MBT requirements together with a couple of sample models. For the benchmark, we made model variations such that we cover different guard and variable types.

We describe 2 out of the 5 Event-B models that we used for the benchmarks. The events of the first one contain numerical parameters, while the events of the second model focus on set parameters. The presentation of each model starts with a short description, followed by the types of the global variables (defined in the context of the Event-B model). Then, the events of the Event-B machines are listed together with their parameters. The guards and actions associated to each event are presented in a separate table.

Numerical-based model: Bank Account. The first example models a simple bank account system. The system allows the user to deposit money in the account or to withdraw money from it. The bank pays interest and charges fees. Depending on the current balance, a deposit can be in four states: overdraft, empty, silver and gold. Thus, the Event-B variables are: $balance \in \mathbb{Z}$, $transaction \in BOOL$ and $state \in$ *STATES={overdraft,empty,silver,gold}*. The machine events, whose guards and actions are given in Table 3.8, are the following:

E1. *Initialization*, that initializes the bank account

E2. *Deposit*, having the numerical parameters $amount1$ and $amount2$

E3. *Withdraw*, having the numerical parameters $amount1$ and $amount2$

E4. *ValidateOverdraft*

E5. *ValidateEmpty*

Table 3.8: Guards and actions of Bank Account events

Ev	Guards	Actions
E1:	*TRUE*	$balance := 0$, $state :=$ *empty* $transaction :=$ *FALSE*
E2:	$amount1 + amount2 > 200 \wedge$ $amount1 \in \mathbb{N} \wedge amount2 \in \mathbb{N}$	$balance := balance + amount1 + amount2$ $transaction :=$ *TRUE*
E3:	$balance > 0 \wedge amount1 + amount2 < 1000 \wedge$ $balance - amount1 - amount2 > -100 \wedge$ $amount1 \in \mathbb{N} \wedge amount2 \in \mathbb{N}t$	$balance := balance - amount1 - amount2$ $transaction :=$ *TRUE*
E4:	$balance < 0 \wedge balance > -100$	$state := overdraft$
E5:	$balance = 0$	$state :=$ *empty*
E6:	$balance > 0 \wedge balance < 1000$	$state :=$ *silver*
E7:	$balance \geq 1000$	$state :=$ *gold*
E8:	$balance > 1500 \wedge$ $value \leq 50 \wedge value > 0 \wedge value \in \mathbb{N}$	$balance := balance + value$
E9:	*fee* $> 0 \wedge$ *fee* $< 50 \wedge$ *fee* $\in \mathbb{N} \wedge$ $transaction =$ *TRUE*	$balance := balance -$ *fee* $transaction :=$ *FALSE*

E6. *ValidateSilver*

E7. *ValidateGold*

E8. *PayInterest*, having the numerical parameter $value$

E9. *ChargeFee*, having the numerical parameter *fee*.

Set-based model: Basket of Items. Here we model a basket of items. The system allows the user to add items, to remove items and to pick items from the basket. The system checks if the basket is empty or full or can make a special check. The global variables are: $items \in \mathbb{P}(\textit{ITEMS})$, *buffer* $\in \mathbb{P}(\textit{ITEMS})$, *isEmpty* $\in BOOL$, *isFull* $\in BOOL$, *CAPACITY* $\in \mathbb{N}$, and $count \in \mathbb{N}$ with the invariants $count \geq 0 \wedge count \leq$ *CAPACITY* and $count = \mathrm{card}(items)$, where *ITEMS* $= \{it1, it2, \ldots, it20\}$. The Event-B events, whose guards and actions are given in Table 3.9, are the following:

E1. *Initialization*, that initializes the basket of items

E2. *PickItems*, with the set parameter its

Table 3.9: Guards and actions of Basket of Items events

Ev Guards	Actions
E1: *TRUE*	$items := \emptyset$, *buffer* $:= \emptyset$, $count := 0$ *CAPACITY* $:=$ card(*ITEMS*) *isEmpty* $:=$ *TRUE*, *isFull* $:=$ *FALSE*
E2: $its \subseteq$ *ITEMS*	*buffer* $:= its$
E3: *buffer* $\subseteq$ *ITEMS* $\wedge$ card(*buffer*) $> 5 \wedge$ card(*buffer*) $+ \, count \leq$ *CAPACITY*	$items := items \cup$ *buffer* $count :=$ card($items \cup$ *buffer*) *isEmpty* $:=$ *FALSE*
E4: *buffer* $\subseteq items \;\wedge$ card(*buffer*) $> 3 \wedge$ $count -$ card(*buffer*) ≥ 0	$items := items \setminus$ *buffer* $count :=$ card($items \setminus$ *buffer*)
E5: $items = \emptyset \wedge count = 0$	*isEmpty* $:=$ *TRUE*
E6: $\{it1, it20\} \subseteq items \wedge \text{card}(items) < 6$	*buffer:=items*
E7: $count =$ *CAPACITY*	*isFull* $:=$ *TRUE* $items := \emptyset$, $count := 0$

E3. *AddItems*

E4. *RemoveItems*

E5. *ValidateEmpty*

E6. *CheckSpecial*

E7. *ValidateFull.*

Mixed types model: numerical and set parameters. This third example an artificially created model that allows us to experiment with different types of events. The global variables are: $setA \in \mathbb{P}(A)$, $setB \in \mathbb{P}(B)$, $value \in \mathbb{N}$, $sizeA \in \mathbb{N}$, $sizeB \in \mathbb{N}$, with the invariants: $sizeA = \text{card}(setA)$ and $sizeB = \text{card}(setB)$, *found* $\in BOOL$ and *full* $\in BOOL$. The events, whose guards and actions are given in Table 3.10, are the following:

E1. *Initialization*

E2. *SearchInA*, with the set parameter $s1$

E3. *ValidateFull*

E4. *SearchInB*, with the set parameter $s1$

E5. *Increase*, with the set parameters $s1, s2$ and the numerical parameters $v1, v2$

E6. *AFullBEmpty*

E7. *BFullAEmpty*

E8. *UpdateAll*, with the set parameters $s1, s2$ and the numerical parameter $v1$.

Table 3.10: Guards and actions of mixed model events

Ev	**Guards**	**Actions**
E1:	*TRUE*	$setA := \emptyset$, $sizeA := 0$ $setB := \emptyset$, $sizeB := 0$ $value := 0$, $\mathit{found} := \mathit{FALSE}$, $\mathit{full} := \mathit{FALSE}$
E2:	$s1 \subseteq setA \land \mathrm{card}(s1) = 2 \land a1 \in s1 \land$ $s1 \in \mathbb{P}(A)$	$\mathit{found} := \mathit{TRUE}$
E3:	$sizeA = \mathrm{card}(A) \land sizeB = \mathrm{card}(B) \land$ $value \geq 100$	$\mathit{full} := \mathit{TRUE}$
E4:	$s1 \subseteq setB \land \mathrm{card}(s1) = 2 \land s1 \in \mathbb{P}(B) \land$ $\{b1, b2\} \subseteq setB$	$\mathit{found} := \mathit{TRUE}$
E5:	$s1 \subseteq A \land s2 \subseteq B \land v1 - v2 < 10 \land$ $v1 \in \mathbb{N} \land v2 \in \mathbb{N}$	$setA := setA \cup s1$, $sizeA := \mathrm{card}(setA \cup s1)$ $setB := setB \cup s2$, $sizeB := \mathrm{card}(setB \cup s2)$ $value := value + v1 + v2$
E6:	$sizeA = \mathrm{card}(A) \land sizeB = 0$	$\mathit{full} := \mathit{TRUE}$
E7:	$sizeA = 0 \land sizeB = \mathrm{card}(B) \land value >$ 0	$\mathit{found} := \mathit{TRUE}$
E8:	$s1 \subseteq A \land s2 \subseteq B \land$ $v1 \in \mathbb{N}$	$setA := s1$, $sizeA := \mathrm{card}(s1)$ $setB := s2$, $sizeB := \mathrm{card}(s2)$ $value := v1$

3.3.3 Test data generation using genetic algorithms

Before describing our test generation approach, let us establish the problem to be solved. First, let us note that Event-B specifications are event-based rather than state-based models. Formally, these are abstract state machines [135] in which the (implicit) states are given by the (global) values of the *variables* on which the events operate. Each event is given by a triplet consisting of (1) the *parameters* (local variables) used by the event, (2) the guards which constrain the event application (the guards may involve both local and global variables) and (3) the actions of event, which may change the values of the global variables. The events produce the transitions between states: the guards establish the valid source state(s) of the transition while the actions produce the target state(s). In general, the application of an event depends on the values of the parameters it receives. If we want to execute a path (sequence of events) through the model, we will need to find appropriate parameter values for each event in the sequence (i.e. which satisfy the corresponding guards). This is the problem we will solve using a genetic algorithm. Naturally, the prerequisite is that a set of paths, which satisfies the given test requirement has already been found.

In general, this requirement is expressed as a level of coverage of the model. Various levels of coverage for finite state machines exist in the literature [134, 148] and some can be adapted to Event-B models without the need to transform the model into an explicit state machine (for large systems this transformation may be impractical). For example, transition coverage for a finite state machine requires every transition to be triggered at least once. Similarly, for Event-B models, event coverage will involve the execution of every event at least once. This type of coverage can be generalized by requiring that each feasible sequence of events of a given length k is executed at least once. Obviously, in order to decide if a path is feasible or not it may be necessary to effectively find test data (parameter values) which triggers it. Consequently, the potentially feasible paths can be selected first by deleting paths which contain obvious contradictory constraints (e.g. both C and $\neg C$) and then the test data generation algorithm is applied to each such path. Other types of coverage may also be defined but this beyond the scope of this section.

In this section, we assume that we have a set of paths (that cover, for instance, all events of the model). For each path of the given set, we seek appropriate test data, i.e. event parameters which enable the events in the path. It may be possible that the test data for the selected path has not been found, either because of the complexity of the guard constraints or simply because the path is infeasible; if this is the case, a new path is selected. Note that the section does not address the issue of path selection, but only the test generation for the chosen path(s).

Below we present the theoretical instruments based on genetic algorithms for the above problem. First, Subsection 3.3.3 provides the background on genetic algorithms. Then, the Subsections 3.3.3 and 3.3.3 describe the main ingredients of the approach, i.e. the encoding of the sought solutions into chromosomes and the fitness function that guides the search into the solution space, respectively.

Note that among the different meta-heuristic algorithms, for convenience, in this section we have chosen to use the class of genetic algorithms [143], because they are widely used in search-based testing approaches and have good tooling support. However, we plan in the future to experiment with other types of algorithms like simulated annealing or particle swarm optimization.

Genetic algorithms

Genetic algorithms (GAs) [143] are a particular class of *evolutionary algorithms*, that use techniques inspired from biology, such as selection, recombination (crossover) and mutation. GAs are used for problems which cannot be solved using traditional techniques and for which an exhaustive search of the solution space is impractical. In particular, the application of GAs to the difficult problem of test data generation recently received an increased attention from the testing research community [141, 140].

GAs basic approach is to encode a population of potential solutions on some data structures, called *chromosomes* (or *individuals*) and applying recombination and mutation operators to these structures. A high-level description of a genetic algorithm [140, 143] is given in Fig. 3.10. The *fitness (or objective) function* assigns a score (called fitness) to each chromosome in the current population. The fitness of a chromosome depends on how close that chromosome is to the solution of the problem. Throughout this section, the fitness is considered to be positive and finding a solution corresponds to minimizing the fitness function, i.e. a solution will be a chromosome with fitness 0. The algorithm terminates when some stopping criterion has been met, for example when a solution is found, or when the number of generations has reached the maximum allowed limit.

Various mechanisms for selecting the individuals to be used to create offspring, based on their fitness, have been devised [137]. GA researchers have experimented with mechanisms such as sigma scaling, elitism, Boltzmann selection, tournament, rank and steady-state selection [143].

After the selection step, recombination takes place to form the next generation from parents and offspring. The mutation operator is then applied. These two operations, crossover and mutation, depend on the type of encoding used and so they are discussed in more detail in the next subsection.

Chromosome encodings

Consider a path $event_1 \dots event_n$ in the Event-B model. A *chromosome* (possible solution) is a list of values, $x = (x_1, \dots, x_m)$ for the event parameters of the path events (in the order they appear). More formally, if $p_{i1}, \dots, p_{ik_i}$ are the parameters of $event_i$,$1 \leq i \leq n$, then x represents a list of values for parameters $p_{11} \dots p_{nk_n}$. Naturally, $m = k_1 + \dots + k_n$ can differ from the number n of events in the sequence. If the values x satisfy all guards and, consequently, trigger the path,

```
Randomly generate or seed initial population $P$
Repeat
    Evaluate fitness of each individual in $P$
    Select $P'$ from $P$ according to selection mechanism
    Recombine parents from $P'$ to form new offspring
    Construct $P'$ from parents and offspring
    Mutate $P'$
    $P \leftarrow P'$
Until Stopping Condition Reached
```

Figure 3.10: Genetic Algorithm

then x is a solution for the given path. For numerical data, the chromosomes are integer-encoded, each gene representing one parameter.

Consider, for example, the path $E2$ $E8$ $E9$ $E3$ $E7$ from the Bank Account example presented earlier (technically, any path of a model starts with the special event *Initialization* ($E1$), but for simplicity when we mention the events of a path we skip $E1$). There are five events in the path: $E2$ (*Deposit*), which receives $amount1$ and $amount2$ as parameters, $E8$ (*PayInterest*), with parameter $value$, $E9$ (*ChargeFee*), with parameter *fee*, $E3$ (*Withdraw*), with parameters $amount1$ and $amount2$ and $E7$ (*ValidateGold*), with no parameters. Since all 6 parameters have numerical types, a chromosome for the above path will be a list of 6 integers.

An additional problem occurs when non-numerical types are involved since such values will have to be encoded into the chromosome. The applications we have considered use enumeration types as well as types derived from these using traditional set operators ($\cup, \setminus, \times$). For a k-valued type $T = \{v_1, \ldots, v_k\}$, a set parameter S which is a subset of T, i.e. $S \subseteq T$, is represented by a bitmap of length k, which has 1 on the ith position in the bitmap if $v_i \in S$, and 0 otherwise. Then, a chromosome corresponding to parameters $p_1 \ldots p_m$ will be a list of values $x_1 \ldots x_m$, in which each value is encoded as appropriate. The applications we have considered use both numerical and non-numerical types and so some values in the chromosome are represented by simple integers whereas other values are encoded as bitmaps. Once we generated a population of chromosomes, the operations of crossover and mutation are applied as described below.

Crossover. For mixed chromosomes (with both binary and integer genes) and binary-only chromosomes, a *single-point crossover* is used. This randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two new offspring. For example, the strings 00000000 and 11111111 could be crossed over at the third locus to produce the two offspring 00011111 and 11100000. The crossover is applied to individuals selected at random, with a probability (rate) p_c. Depending on this rate, the next generation will contain the parents or the offspring.

For integer chromosomes, we used a *heuristic real value crossover*, inspired from [142], that showed to be the most efficient type of crossover for our problem. This uses the fitness of the individual for determining the direction of the search. For parents $x = (x_1, \ldots, x_m)$, $y = (y_1, \ldots, y_m)$, x fitter than y, one offspring $z = (z_1, ..., z_m)$ is generated, with z_i being the integer-rounded value of $\alpha \cdot (x_i - y_i) + x_i$, $\alpha \in (0, 1)$. Heuristic real value and single-point crossovers can be combined.

Mutation is used to introduce variation in the population by randomly changing genes in the chromosome. Mutation can occur at each bit position in a string with some probability p_m, usually very small [143]. For binary genes, the mutation operator randomly flips some bits in a chromosome. For example, the string 00000100 could be mutated in its second position to yield 01000100. For integer genes, the gene value is replaced by another integer value that is randomly chosen from the same interval.

Fitness function

The algorithm evaluates a candidate solution by executing each event with the values encoded in the chromosome's genes until the guard of the current event is not satisfied. The fitter individuals are the ones which enable more events from the given path. They are rewarded with a lower fitness value. The fitness function is calculated using a formula widely used in the search-based testing literature [141, 139], using two components. The first evaluates how close a chromosome is to executing the given path, by counting the events executed. The second measures how close is the first unsatisfied guard predicate to being true.

$$\textit{fitness} := approach\_level + normalized\_branch\_level$$

The first component, *approach (approximation) level* is similar a metric in evolutionary structural test data generation [141]. This is calculated by $m - 1 - n$, where m is the length of the path to be executed and n is the number of events successfully executed until the first unsatisfied guard on the path, as in Fig. 3.11.

A fitness function formed only from the approach level has many plateaux (i.e. for each value $0, 1, \ldots, m - 1$) and it would not offer enough guidance to the search. Consequently, the second component, called *branch level*, was introduced. This computes, for the place where the actual path diverges from the required one, how close was the guard predicate to being true.

For numeric types, the *branch level* can be derived from the guards predicates using Tracey's objective functions as shown in Table 3.11 [147, 140]. The *branch level* is then mapped onto the interval [0,1) or normalized.

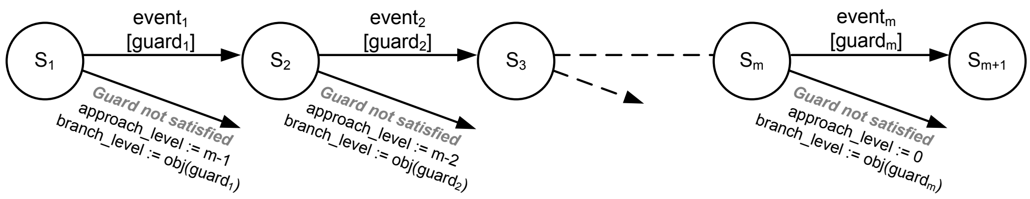


Figure 3.11: Calculating the fitness function

Table 3.11: Tracey's objective functions for relational predicates and logical connectives. The value K, $K > 0$, refers to a constant which is always added if the term is not true.

Relational predicate or logical connective	Objective function obj
$a = b$	if $abs(a-b) = 0$ then 0 else $abs(a-b) + K$
$a \neq b$	if $abs(a-b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a-b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a-b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b-a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b-a) + K$
Boolean	if *TRUE* then 0 else K
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$
a xor b	$obj((a \wedge \neg b) \vee (\neg a \wedge b))$
$\neg a$	Negation is moved inwards and propagated over a

We extended the calculation of the *branch level* to applications which involve set theory based constraints as described below. The applications considered use basic types that can be mapped onto either an interval ($[p..q]$, $0 \leq p < q$,) or an enumeration of non-negative integers ($\{p_1, \ldots, p_n\}$, $n \geq 1$, $p_i \geq 0$, $1 \leq i \leq n$). Furthermore, the derived types use the $\cup$, $\cap$, $\setminus$ and $\times$ set operators. Then the objective function for the $a \in A$ and $a \notin A$ predicates can be derived using the transformations given at the top of Table 3.12. The formulae are then extended for the $\subseteq$ and $=$ set operators, as shown at the bottom of Table 3.12.

Table 3.12: The extension of Tracey's objective functions to set operators

Predicate involving $\in$ for basic or derived sets	Objective function obj
$a \in [p, q]$	$obj((a \geq p) \wedge (a \leq q))$
$a \notin [p, q]$	$obj((a < p) \vee (a > q))$
$a \in \{p_1, \ldots, p_n\}$	$obj((a = p_1) \vee \cdots \vee (a = p_n))$
$a \notin \{p_1, \ldots, p_n\}$	$obj((a \neq p_1) \wedge \cdots \wedge (a \neq p_n))$
$a \in A \cup B$	$obj((a \in A) \vee (a \in B))$
$a \in A \cap B$	$obj((a \in A) \wedge (a \in B))$
$a \in A \setminus B$	$obj((a \in A) \wedge (a \notin B))$
$(a, b) \in (A, B)$	$obj((a \in A) \wedge (b \in B))$
Predicates for $\subseteq$ and $=$ operators	Objective function obj
$[p, q] \subseteq A$	$obj(\wedge_{i=p}^{q}(i \in A))$
$\{p_1, \ldots, p_n\} \subseteq A$	$obj((p_1 \in A) \wedge \cdots \wedge (p_n \in A))$
$[p, q] \nsubseteq A$	$obj(\vee_{i=p}^{q}(i \notin A))$
$\{p_1, \ldots, p_n\} \nsubseteq A$	$obj((p_1 \notin A) \vee \cdots \vee (p_n \notin A))$
$A = B$	$obj((A \subseteq B) \wedge (B \subseteq A))$
$A \neq B$	$obj((A \nsubseteq B) \vee (B \nsubseteq A))$

3.3.4 Experiments

We have implemented our approach as a plugin for the Eclipse-based Rodin platform for Event-B. The plugin is designed to automatically generate test data for given paths in the Event-B model. It can generate test data, i.e. the input parameters for the events on the given path, employing the fitness function described in Section 3.3.3. The execution of the events (including the initialization) was performed using the Event-B model simulation of the ProB plugin [131].

For the benchmark of 5 models mentioned in Section 3.3.2, we have considered a set of 18 random paths likely to be feasible, which covered all the events from the models. The paths length varied between 2 and 5 events (without counting the initialization event). The number of parameters on each path varied between: (a) 1 and 7 for numerical models; (b) 1 and 2 non-numerical parameters, such as $x \in \mathbb{P}(\mathit{ITEMS})$ for set examples; (c) $2-4$ set parameters and $2-3$ numerical parameters for the mixed model. The codification used was: integer-valued for numerical parameters (the integer range was fixed to 2000) and bitmap for set parameters.

As recommended in [133], a search algorithm (GA in this case) should be compared with random search in order to check that the algorithm is not simply successful because the search problem is easy. Therefore, we tried to generate test data for the 18 selected paths mentioned above, denoted by $P1 - P18$, using the two methods: *search-based testing with genetic algorithms (GA)* and *random testing (RT)*. For each path and each test generation method, 30 runs were performed (this number was also recommended in [133]). A run is considered *successful* if it can produce input test data that can trigger the whole path, or equivalently, the fitness function associated has the value 0. The run ends when a solution was found or when the maximum number of generations was reached.

Using genetic algorithms, the amount of time needed to obtain test data for a path, i.e. the actual values of the parameters which trigger the path, varied between 1 second (for very simple paths, where the solution could be found from the first generation) and 60 seconds (for complex paths).

The genetic algorithm framework used for experimentation was the open source Java Genetic Algorithms Package (JGAP) [138]. The maximum number of generations for the genetic algorithm was set to 100 and the population size to 20. The selection operator employed was *BestChromosomesSelector*, an elitist operator, the mutation rate was $p_m = 1/10$ and the crossover was single-point (for non-numerical parameters) or heuristic crossover (for numerical ones), as presented in Section 3.3.3.

For random testing, the same library was used: instead of applying recombination or mutation, the population was randomly generated at each step, ensuring this way an equal treatment, i.e. an equal number of generations (or fitness function evaluations) for both methods, GA and RT. For each run, the generation when the solution was found was recorded and Table 3.13 presents the summarizing data: the success rate for each method (percent of successful runs from the 30 ones considered) and other descriptive statistics, e.g. the average (mean) number of generations, the median and the standard deviation.

Statistical tests should be realized to support the comparison of GA and RT runs. In our experiments we have used two statistical tests: the parametric t-test and the non-parametric Mann-Whitney U-test. The null hypothesis (H_0) is thus formulated as follows: *There is no difference in efficiency (the number of generations needed to find a solution) between GA and RT.* The alternative hypothesis (H_a) follows: there is a difference between the two approaches, GA and RT. The two tests measure different aspects: the t-test measures the difference in mean values (the null hypothesis is $H_0 : \mu_1 = \mu_2$), whereas the Mann-Whitney U-test measures their difference in medians ($H_0' : \theta_1 = \theta_2$), i.e. whether the observations in one data sample are more likely to be larger than observations in the other sample.

The test results and the p-values obtained are given in Table 3.13. In the columns t-test and U-test, the sign '$+$' stands for rejecting the null hypothesis (consequently, there is a statistically significant difference between GA and RT results), while the '$-$' indicates that the null hypothesis cannot be rejected at the significance level considered, $\alpha = 0.01$. The p-value computed by the statistical test is also provided, excepting the case when it can not be computed, e.g. when both approaches were able to find a solution from the first generation for all the runs (paths $P16, P17$), where '†' stands for not computed.

Some standardized effect size measures were also used and they are given in the last two columns: the Vargha and Delaney's A statistic, the Cohen's D coefficient. The Vargha and Delaney's A statistic [133] is a performance measure, used to quantify the probability that GA yield 'better values' than RT. In our case, 'better values' means lower number of generations needed to obtain a solution.

The Vargha and Delaney's statistics is given in the column 'A'. For simple paths, where RT and GA provide the solution in the same number of generations, the effect size is 0.5. For more complex paths, a value of 0.82 means that we would obtain better results in 82% of the time with GA (they guide the search to success in a lower number of generations). It is worth noting that *GA clearly outperformed RT for* 14 *out of* 18 *paths considered*, and the difference in terms of success rate, average (or median) number of generations was significant.

The last column of Table 3.13 presents the Cohen's D coefficient, which is computed as the absolute difference between two means, divided by a pooled standard deviation of the data [133, 145]. According to [145], Cohen has proposed the following 'D values' as criteria for identifying the magnitude of an effect size: a) small effect size: $D \in (0.2, 0.5)$, b) medium effect size $D \in [0.5, 0.8)$, c) large

Table 3.13: Success rates, results of the statistical tests and effect size measures

Path	Meth.	Success	Avg.	Median	Std.	t-test	U-test	A	D
		rate	gen.		dev.	p-val	p-val		
P1	GA	100.0%	18.2	12.0	18.8	+	+	1.00	5.85
P1	RT	3.3%	99.0	100.0	5.3	< 0.001	< 0.001		
P2	GA	100.0%	14.9	11.0	12.5	+	+	1.00	6.45
P2	RT	3.3%	97.6	100.0	13.1	< 0.001	< 0.001		
P3	GA	96.7%	22.7	14.5	19.8	+	+	0.98	4.61
P3	RT	10.0%	96.9	100.0	11.1	< 0.001	< 0.001		
P4	GA	100.0%	12.4	8.0	11.6	+	+	0.82	1.17
P4	RT	96.7%	35.0	32.5	24.8	< 0.001	< 0.001		
P5	GA	66.7%	53.3	44.5	38.7	+	+	0.83	1.71
P5	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P6	GA	100.0%	23.5	21.0	9.2	+	+	1.00	11.73
P6	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P7	GA	100.0%	12.7	12.0	4.5	+	+	1.00	16.72
P7	RT	6.7%	98.5	100.0	5.7	< 0.001	< 0.001		
P8	GA	100.0%	16.9	17.0	4.4	+	+	1.00	26.59
P8	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P9	GA	100.0%	13.7	13.0	2.4	+	+	1.00	12.46
P9	RT	3.3%	98.3	100.0	9.3	< 0.001	< 0.001		
P10	GA	100.0%	30.9	31.5	6.3	+	+	1.00	15.51
P10	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P11	GA	96.7%	20.0	13.0	22.6	+	+	0.98	4.64
P11	RT	3.3%	98.6	100.0	7.9	< 0.001	< 0.001		
P12	GA	100.0%	13.5	13.0	2.8	+	+	1.00	43.66
P12	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P13	GA	100.0%	11.9	11.5	2.2	+	+	1.00	56.56
P13	RT	0.0%	100.0	100.0	0.0	< 0.001	< 0.001		
P14	GA	100.0%	1.3	1.0	1.6	–	–	0.47	0.29
P14	RT	100.0%	1.0	1.0	0.0	0.28	0.49		
P15	GA	100.0%	1.0	1.0	0.0	–	–	0.50	†
P15	RT	100.0%	1.0	1.0	0.0	†	1.00		
P16	GA	100.0%	1.0	1.0	0.0	–	–	0.50	†
P16	RT	100.0%	1.0	1.0	0.0	†	1.00		
P17	GA	90.0%	16.9	5.5	29.9	+	+	0.91	1.79
P17	RT	53.3%	72.2	83.5	31.7	< 0.001	< 0.001		
P18	GA	100.0%	1.8	1.0	2.4	–	–	0.53	0.17
P18	RT	100.0%	1.5	1.0	0.8	0.53	0.63		

effect size $D \in [0.8, \infty)$. According to this classification, it can be easily noticed that the *difference between the results obtained with GA versus RT correspond for most paths (*14 *out of* 18*) to a large effect size*.

3.3.5 Final discussion

Bottom line. In this section, we have presented an approach based on genetic algorithms that allows generating test data for event paths in the Event-B framework. One distinguishing feature of Event-B is its set-theoretic foundation, meaning that in Event-B models, numerical variables are used together with non-numerical types based on sets. To address this, we extended the fitness functions available in the search-based testing literature to set types. Moreover, the encoding of the sought solutions included mixed chromosomes containing both numerical and non-numerical types. Finally, we followed standard statistical guidelines [133] to demonstrate the efficiency and effectiveness of our implementation on a diversified benchmark inspired by discussions with the industry.

Related work. The only approach of test generation for Event-B models is based on explicit model-checking [80] with ProB [131], which suffers from the classical state space explosion problem. There is also related work on applying search-based techniques to EFSMs [139, 136, 149]. Differently from these, we address a different modeling language and tackle non-numerical types. However, we can certainly extend our work with ideas from these papers, e.g. regarding feasible path generation, or from previous work on test generation from B models (the precursor of Event-B language, even though B is not an event-based language) [144], [148, ch.3].

Future work. Since the goal is to develop a test method that scales for industrial Event-B models, we have performed a survey of 29 publicly available Event-B models posted by the DEPLOY academic and industrial partners[11]. Beside the large size of industrial models, there are a couple of other dimensions still to be addressed. For instance, Event-B uses a rich set of operations as well as complex data based on set relations, sets of sets or partial functions. In principle, these can be mapped to sets and use the proposed methods but this may not scale, so the fitness functions and encodings might need to be further specialized for these operators. Moreover, industrial models are usually decomposed in order to mitigate modeling complexity, which means that we have to extend our methods to work for modular and component-based models.

[11] *http://deploy-eprints.ecs.soton.ac.uk/view/type/rodin=5Farchive.html*

3.4 Test suite optimisations

Multi-objective test suite reductions Large test suites generated by automatic test generators usually need to be optimised according to different criteria. In the following, we follow a more general approach of defining the reduction criteria as multi-objective test suite optimization problems. They are solved using two modern Multi-Objective Evolutionary Algorithms, namely: NSGA-II [121] and SPEA-2 [129]. The experiments have been conducted using five test suites generated from two industrially-inspired Event-B models.

The rest of this section is structured as follows. We introduce the test suite optimization problem for Event-B models, then we mathematically define the six different test suite optimizations, and we finally describe the experiment setup and results.

The presented results are based on [132] and have used publicly available Event-B models rather than the SAP internal models.

3.4.1 Multi-Objective Test Suite Optimization

Given an Event-B machine M with $E = \{e_1, e_2, ..., e_m\}$ the set of its events, a test case can be defined as a sequence of events in E that can be executed in the machine M (an execution path). Each test case begins with a special event called *INITIALISATION* which serves to initialize the global variables of the machine before starting the execution of a test case. A test suite is by definition a collection of test cases.

We introduce the multi-objective test suite minimization problem. We adopt here the definitions from [128]. Generally, a multi-objective optimization problem can be defined as to find a vector of decision variables x, which optimizes a vector of M objective functions $f_i(x), 1 \leq i \leq M$. The objective functions are the mathematical formulations of the optimization criteria. Usually, these functions are conflicting, which means that improvements with respect to one function can only be achieved when impairing the solution quality with respect to another objective function. Solutions that can not be improved with respect to any functions without impairing another one are called *Pareto-optimal solutions*.

Formally, let us assume that, without loss of generality, the goal is to minimize the functions $f_i(x), 1 \leq i \leq M$. A decision vector x is said to *dominate* a decision vector y (we write $x \succ y$) if and only if the following property is satisfied by their objective vectors:

$$f_i(x) \leq f_i(y), \forall i \in \{1, 2, ..., M\} \text{ and } \exists i_0 \in \{1, 2, ..., M\}, f_{i_0}(x) < f_{i_0}(y).$$

The dominance relations states that a solution x is preferable to another solution y if x is at least as good as y in all objectives *and* better with respect to at least one objective. The *Pareto-optimal set* is the set of all decision vectors that are not dominated by any other decision vectors. The corresponding objective vectors are said to from *Pareto frontier*. Therefore, the multi-objective optimization problem can be defined in the following manner:

Given: a vector of decision variables, x, and a set of objective functions, $f_i(x), 1 \leq i \leq M$,

Problem: minimize$\{f_1(x), f_2(x), ..., f_M(x)\}$ by finding the Pareto-optimal set over the feasible set of solutions.

With respect to multi-criteria test suite optimization, the objective functions f_i are the mathematical descriptions of the testing criteria that must be satisfied to provide desired adequate testing of the model. In real industrial testing problems, there exist multiple test criteria, because a single ideal criterion is simply impossible to be achieved. For example, a frequently optimization problem is to produce a minimal test suite which achieves maximal coverage of the model entities with a minimal execution cost. Therefore, this is a *bi-objective minimization* test suite problem.

Formally, multi-objective test suite optimization problem can be defined in the following manner [128]:

Multi-Objective Test Suite Optimization.

Given: a test suite TS, a vector of M objective functions $f_i, 1 \leq i \leq M$

Problem: to produce a subset $T \subset TS$, such that T is a Pareto-optimal set with respect to the set of the above objective functions.

In the following, we instantiate this general multi-objective test suite optimization problem with respect to our Event-B models.

We assume an Event-B machine M for which we have generated a test suite TS. Of course, TS satisfies a set of test requirements which are expressed as a level of coverage of the model. For the moment, we only consider that the test suite TS achieves the following simple coverage criterion:

Event Coverage Criterion: A test suite $TS = \{t_1, ..., t_m\}$ of m test cases for an Event-B model M is said to achieve *event coverage criterion* if and only if for each event e of the model M there exists a test case $t_i \in TS$ which covers e.

Having the above criterion in mind, we can formulate the following optimization problem:

Test Suite Minimization Problem.

Given: A test suite TS generated for a machine M with $E = \{e_1, e_2, ..., e_n\}$ the set of events, and subsets of TS, T_is, one associated with each of the e_is such that any one of the test cases t_j belonging to T_i can be used to cover e_i.

Problem: Find minimal test suite T from TS which covers all e_i.

As also mentioned in the previous section, this problem is NP-complete because it can be reduced to the minimum set-cover problem [120] as follows. We recall that for us a test case $tc \in TS$ is an execution path which consists in a sequence of events from E. Let be $cov(tc) = \{e \in E|tc \text{ covers } e\}$ the set of events covered by test case tc. By definition, $cov(tc)$ is a subset of E. Therefore the solution T of the above test suite minimization problem is exactly a minimum set cover for E, because

$$\bigcup_{t \in T} cov(t) = E$$

and T is the minimal subset of TS which covers E.

Many solutions have been proposed to solve this test suite minimization problem. [119, 124, 117, 126, 118]. Due to its exponential complexity, we use Multi-Objective Evolutionary Algorithms for solving it. For that, we mathematically reformulate it as a constraint bi-objective test suite optimization problem (see TSO1 problem below).

3.4.2 Optimization Criteria

Based on practical experience at SAP, we propose here different test suite optimization criteria.

TSO1-Minimizing the size of the test suite. Due to the restrictions of time, obtaining a minimal test suite which achieves maximal level of coverage is of particular interest among testers. Therefore the goal of this problem is to produce a test suite that contains the smallest possible number of test cases that achieve the same coverage (in our case, the event coverage) as the complete test suite. We formulate this problem as a constraint bi-objective optimization problem: maximize event coverage (the first objective) by a minimum number of test cases (the second objective) under the constraint that at least a test case has been selected. The problem can be mathematically described in the following manner.

Let be $TS = \{t_1, t_2, ..., t_m\}$ the initial set of m test cases and $E = \{e_1, e_2, ..., e_n\}$ the set of the events to be covered. We recall that $cov(tc)$ is the set of events covered by the test case tc. Given an order between the elements of a set, a subset $T \subset TS$ can be mathematically represented by a binary vector $x = (x_1, x_2, ..., x_m) \in \{0, 1\}^m$ with

$$x_i = \left\{ \begin{array}{ll} 1, & t_i \in T \\ 0, & t_i \notin T \end{array} \right., 1 \leq i \leq m.$$

Therefore the constraint bi-objective test suite optimization problem to be solved is the following:

$$\text{Minimize } (f_1(x), f_2(x))$$

Subject to:

$$\sum_{i=1}^{m} x_i \geq 1 \ (T \neq \emptyset)$$

Where:

$$f_1(x) = 1 - \sum_{i=1}^{m}(x_i \cdot \frac{|cov(t_i)|}{n}) \text{ (maximize the coverage)}$$

$$f_2(x) = \frac{\sum_{i=1}^{m} x_i}{m} \text{ (minimize the size of test suite).}$$

A Pareto-optimal solution of the above problem corresponds to a minimal subset of the test suite TS which achieves a maximal level of coverage. More, we can see that $f_1 : \{0,1\}^m \to [0,1)$ and $f_2 : \{0,1\}^m \to (0,1]$. Therefore we avoid to select the empty set as a solution.

TSO2-Minimizing the number of the executed events. In order to reduce the effort of the testing process, the number of executed events from the whole test suite should be minimized. Therefore we want to obtain test suites which achieve the event coverage criterion with a minimum number of executed events. The first objective function f_1 and the constraint from the problem TSO1 remain valid. Let be $len(tc)$ the length of the test case $tc \in TS$. The second objective function f_2 which can be used to minimize the number of executed events by the subset $T \subset TS$ is

$$f_2(x) = \frac{1}{\sum_{k=1}^{m} len(t_k)} \sum_{i=1}^{m}(x_i \cdot len(t_i)).$$

TSO3-Minimizing the length of the longest execution path. The longer execution paths are harder to maintain. In this problem we control the lengths of the execution paths by minimizing the length of the longest test case. The mathematical formulation is the following:

$$\text{Minimize } (f_1(x), f_2(x))$$

Where $f_1(x)$ is the same as for TSO1 problem and

$$f_2(x) = \max\{len(t_i) | x_i = 1 \text{ and } 1 \le i \le m\}.$$

The second objective function f_2 is used for minimizing the length of the longest test case.

TSO4-Minimizing the execution time. We measure the execution time for each test case tc from the initial test suite TS. Let us denote by $time(tc)$ the execution time of tc. Then the execution time of a test suite $T \subset TS$ is $\sum_{tc \in T} time(tc)$. In this problem the goal is to minimize the execution time of the test suites. The first objective and the constraint are the same as for TSO1 problem. The second objective function f_2 to be minimized is

$$f_2(x) = \sum_{i=1}^{m}(x_i \cdot time(t_i)) \text{ (minimize the execution time).}$$

TSO5-Maximizing the distribution quality. In order to understand the problem proposed here, let us consider a simple example. Let be $T_1 = \{e_1e_3e_4, e_1e_2, e_3e_2e_5\}$ and $T_2 = \{e_2e_2e_4, e_1e_2, e_3e_5\}$ two test suites which cover the set of events $E = \{e_1, e_2, ..., e_5\}$. The events e_1 and e_2 are executed an equal number of times in T_1, while they are not in T_2. We say that T_1 has a better *distribution quality*. Therefore the goal is to obtain test suites with a good distribution of the events. This property is a practical requirement of users.

In the following, we propose an objective function which measures the distribution quality of a given test suite $T \subset TS$. Let be $TS = \{t_1, t_2, ..., t_m\}$ the initial test suite and $E = \{e_1, e_2, ..., e_n\}$ the set of the events. Let be a matrix A which captures the events covered by each test case tc in TS; the number of rows of A equals the number of events to be covered, n, and the number of columns equals the number of test cases in the initial test suite, m. Therefore the entries $(a_{ij})_{1 \leq i \leq n, 1 \leq j \leq m}$ of A are

$$a_{ij} = \begin{cases} k, & t_j \text{ covers } e_i \text{ by } k \text{ times} \\ 0, & e_i \text{ is not covered by } t_j \end{cases}, 1 \leq i \leq n, 1 \leq j \leq m.$$

Let be $x = (x_1, x_2, ..., x_m) \in \{0, 1\}^m$ the mathematical representation of the test suite $T \subset TS$. We define the matrix $D(x)$ to be

$$D(x) = A \times \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_m \end{pmatrix}$$

More exactly, $D(x)$ is a vector of n components $d_i(x), 1 \leq i \leq n$. From the definition, the entry $d_i(x) = \sum_{k=1}^{m}(a_{ik} \cdot x_k)$ of D denotes the number of times the event e_i was covered by the test suite T.

Now the mean amount of executions per event in T is exactly

$$m_T(x) = \frac{1}{n}\sum_{i=1}^{n} d_i(x).$$

If the test suite T has a good distribution of the events, we would expect $d_i(x), 1 \leq i \leq n$ values to stay near the mean value $m_T(x)$. Therefore in order to obtain a good distribution of the events we define the objective function to be minimized in the following manner:

$$f(x) = \frac{1}{n}\sum_{i=1}^{n}(d_i(x) - m_T(x))^2.$$

Let us illustrate this definition on our simple example. We consider that $TS = T1 \cup T_2 = \{e_1e_3e_4, e_1e_2, e_3e_2e_5, e_2e_2e_4, e_1e_2, e_3e_5\}$. Then, $x_1 = (1, 1, 1, 0, 0, 0)$

and $x_2 = (0,0,0,1,1,1)$ are the mathematical descriptions of T_1 and T_2 respectively. Given that, the matrix A will be

$$A = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 2 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

and

$$D(x_1) = A \times \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 1 \\ 1 \end{pmatrix},\ D(x_2) = A \times \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Further calculation shows that $f(x_1) = 0.24$ and $f(x_2) = 0.64$. Therefore the test suite $T1$ has a better distribution of the events.

We formulate this problem as a constraint single-objective optimization problem and search for solutions which minimize $f(x)$ subject to

$d_i(x) \geq 1,\ 1 \leq i \leq n$ (each event is covered at least one time).

TSO6-Balancing the lengths while minimizing the longest path. Finally, we propose here to balance the lengths of the execution paths while we keep valid the two objectives of **TSO3** problem (achieve event coverage while minimize the length of the longest path). Therefore this problem is a 3-objective test suite optimization problem. We search here for test suites which achieve event coverage by short and balanced execution paths. The third objective function can be mathematically formulated as below.

We remember that $len(tc)$ denotes the length of the test case tc. Let be $T \subset TS$ a test suite and x its mathematical description. First, we define the mean of the lengths as

$$m_T^{len}(x) = \frac{1}{|T|}\sum_{i=1}^{m}(x_i \cdot len(x_i)).$$

If the test suite T contains balanced execution paths, the $len(tc), tc \in T$ values will stay near the mean value $m_T^{len}(x)$. Given that, the third objective function to be minimized can be defined as

$$f_3(x) = \frac{1}{|T|}\sum_{i=1}^{m}(x_i \cdot (len(t_i) - m_T^{len}(x))^2)$$

Table 3.14: Summarize the six test suite optimization problems.

Problem	Type	Constraint	Description
TSO1	bi-objective	yes	Minimizing the size of the test suite
TSO2	bi-objective	yes	Minimizing the no. of the executed events
TSO3	bi-objective	no	Minimizing the longest execution path
TSO4	bi-objective	yes	Minimizing the execution time
TSO5	single-obj.	yes	Maximizing the distribution quality
TSO6	3-objective	no	Balancing the lengths + TSO3 problem

We solve all these six test suite optimization problems using multi-objective evolutionary algorithms. In Table 3.14 we summarize the properties of our problems.

3.4.3 Experiments

We provide now the results of a couple of experiments to verify the efficiency and effectiveness of the presented methods.

Solution Encodings. We chose two modern and widely used Pareto efficient genetic algorithms, NSGA-II and SPEA-2 [129]. When using evolutionary algorithms for solving a multi-objective test suite optimization problem, we must properly encode the possible solutions of the problem. Let be $T \subset TS$ a subset of the initial test suite $TS = \{t_1, t_2, ..., t_m\}$. We use the mathematical representation $x \in \{0, 1\}^m$ of T (see Section 3.4.2) to encode the possible solutions. Therefore binary encoding is considered to be a natural representation for the possible solutions. The inclusion and exclusion of a test case within a subset of the initial test suite are represented by 1 and 0 respectively in a binary string (*chromosome* string).

Subjects. We conducted the experiments with a total of five test suite subjects of varying sizes and complexity levels. The test suites were generated from two industrial inspired Event-B models: the BepiColombo and SSFPilot models which are publicly available DEPLOY model repository[12]. The first four machines are different levels of refinements of BepiColombo project and the last machine is the high level of abstraction of SSFPilot model. The sizes of the machines are listed in Table 3.15. Moreover, the test suite generated from these Event-B models were obtained using our MBT plugin[13] available for Rodin. The test generation algorithm is the one presented in the previous section.

The two Event-B models are summarized below:

[12] `http://deploy-eprints.ecs.soton.ac.uk`

[13] `http://wiki.event-b.org/index.php/MBT_plugin`

Table 3.15: Sizes of five test suite subjects generated from two industrial inspired models (number of events, size of test suites and maximum length of test cases).

Subject	No. of events	Size of TS	Max. size of tcs
BepiColombo_M0	5	40	7
BepiColombo_M1	10	170	7
BepiColombo_M2	12	256	7
BepiColombo_M3	16	240	7
SSFPilot_TCTM	13	786	8

- *BepiColombo*: This is an abstract model[14] of two communication modules in the embedded software on a space craft. The Event-B model was proposed for formal validation of software parts of BepiColombo mission to Mars[15]. The model has different levels of refinements. In the abstraction, M_0, the main goal of the system is modeled. The details of the system are added through three refinement levels, M_1, M_2 and M_3. The modeling approach starts on the first level with 5 set-type variables and 5 events and ends up with 18 variables and 16 events.

- *SSFPilot*: This is an Event-B model[16] of a pilot for a complex on-board satellite mode-rich system: Attitude and Orbit Control System (AOCS). In [125] the authors present a formal development of an AOCS in Event-B modeling language. They show that refinement in Event B provides the engineers with a scalable formal technique that enables both development of mode-rich systems and proof-based verification of their mode consistency.

Results. The test suite optimization techniques attempt to reduce the test suite cost w.r.t. a given coverage criterion (event coverage in our case). Given that, the percentage reduction will be used as a measure for comparative analysis. To increase the confidence, we compare the results produced by the two algorithms: NSGA-II and SPEA-2.

We have used the multi-objective evolutionary algorithm framework jMetal [122] for our experiments. The two algorithms were configured with population size of 100. The archive size of SPEA-2 was set to the same value, 100. The stopping criterion is to reach the maximum number of generation which was set to 100. The both algorithms use the following genetic operators: *the binary tournament selection* operator, *the single point crossover* operator with probability of

[14] http://eprints.ecs.soton.ac.uk/22048/5/Rodin_Space_Craft.zip

[15] See http://deploy-eprints.ecs.soton.ac.uk/72/1/BepiColombo_-_Modelling_Approach.pdf and http://en.wikipedia.org/wiki/BepiColombo

[16] http://deploy-eprints.ecs.soton.ac.uk/58/

Table 3.16: TSO1. Average reduced sizes for optimized test suite T.

		NSGA-II		SPEA-2	
Subject	$f_2(x_{TS})$	Avg $f_2(x_T)$	Avg%	Avg $f_2(x_T)$	Avg%
BepiColombo_M0	40	1.03	97.42	1.01	97.47
BepiColombo_M1	170	7.59	95.53	8.72	94.87
BepiColombo_M2	256	28.87	88.72	30.98	87.89
BepiColombo_M3	240	26.14	89.10	27.97	88.34
SSFPilot_TCTM	786	228.42	70.93	232.5	70.41

Table 3.17: TSO2. Average reduced number of executed events for optimized test suite T.

		NSGA-II		SPEA-2	
Subject	$f_2(x_{TS})$	Avg $f_2(x_T)$	Avg%	Avg $f_2(x_T)$	Avg%
BepiColombo_M0	252	8.02	96.8	8.02	96.8
BepiColombo_M1	1300	65.09	94.99	71.93	94.46
BepiColombo_M2	1977	224.42	88.65	236.34	88.04
BepiColombo_M3	1873	204.77	89.06	221.39	88.17
SSFPilot_TCTM	6554	1897.79	71.04	1931.98	70.52

0.9 and *the single bit-flip mutation* operator with the mutation rate of $1/m$ where m is the length of the bit-string (i.e. the size of the initial test suite).

For each test suite subject, each optimization problem and each algorithm, 100 independent runs were performed. The results are presented in Tables 3.16-3.21. To compare the results, we computed for each problem the specific objective function values for the initial test suite. For example, the column $f_3(x_{TS})$ from the Table 3.21 indicates the values of the third objective function of the problem TSO6 when computed for the initial test suite TS. Otherwise, in each table, the average values of specific objective functions of the solutions are indicated. As shown in the tables, the results of the two algorithms are comparable. We obtained high values for the percentage reduction of test suite because of the simplicity of the event coverage criterion.

3.4.4 Conclusions

In this subsection the multi-objective test suite optimization problem for Event-B testing was introduced. Different optimization criteria were proposed and the resulted problems were solved using two modern multi-objective evolutionary algorithms. For all optimization problems the considered test adequacy criterion was the event coverage. All our optimization problems can be easily formulated

Table 3.18: TSO3. Average length of the longest path of optimized test suite T.

Subject	NSGA-II	SPEA-2
BepiColombo_M0	4.69	4.84
BepiColombo_M1	7	7
BepiColombo_M2	7	7
BepiColombo_M3	7	7
SSFPilot_TCTM	8	8

Table 3.19: TSO4. Average execution time (in seconds) of optimized test suite T.

		NSGA-II		SPEA-2	
Subject	$f_2(x_{TS})$	Avg $f_2(x_T)$	Avg%	Avg $f_2(x_T)$	Avg%
BepiColombo_M0	4.6	0.13	97.07	0.14	96.95
BepiColombo_M1	48.43	1.88	96.11	2.16	95.54
BepiColombo_M2	130.16	12.39	90.48	13.40	89.70
BepiColombo_M3	204.28	20.43	89.99	22.13	89.16
SSFPilot_TCTM	197.80	50.78	74.32	51.38	74.02

Table 3.20: TSO5. Average distribution quality of optimized test suite T.

		NSGA-II		SPEA-2	
Subject	$f(x_{TS})$	Avg $f(x_T)$	Avg%	Avg $f(x_T)$	Avg%
BepiColombo_M0	520.24	0.16	99.96	0.16	99.96
BepiColombo_M1	8771.4	17.03	99.80	22.45	99.74
BepiColombo_M2	19840.90	238.98	98.79	270.26	98.63
BepiColombo_M3	14432.43	169.14	98.82	191.42	98.67
SSFPilot_TCTM	166187.40	13251.76	92.02	13667.67	91.77

Table 3.21: TSO6. Average balancing values of the lengths of optimized test suite T.

		NSGA-II		SPEA-2	
Subject	$f_3(x_{TS})$	Avg$f_3(x_T)$	Avg%	Avg$f_3(x_T)$	Avg%
BepiColombo_M0	2.16	0.00	100	0.00	100
BepiColombo_M1	1.81	0.21	88.27	0.22	87.52
BepiColombo_M2	1.57	0.33	78.41	0.34	77.87
BepiColombo_M3	1.62	0.36	77.76	0.37	77.04
SSFPilot_TCTM	2.21	1.15	47.96	1.17	47.05

in a more general framework: a test suite T must meet a set of n requirements $\{r_1, r_2, ..., r_n\}$ to provide the desired 'adequate' testing of the model. We will consider in the future more complex coverage criteria.

3.5 Tool description

In this section we provide more details on the Rodin plug-in implementing the previous introduced methods. A model for BepiColombo is used as a running example. At the end a couple of screenshots are made available. Please check out also the webpage of the plugin at:
http://wiki.event-b.org/index.php/MBT_plugin.

3.5.1 Tool overview driven by the BepiColombo example

In this section we describe the main features of our tool and exemplify it using an Event-B model for the BepiColombo use case, introduced in [130].

Overview. The bird's eye view of the tool is depicted in Fig. 3.12. We take as input an Event-B model M and a finite bound ℓ and we output a finite cover automaton approximating the set of feasible sequences of events of M of length up to ℓ and a test suite, i.e. a set of sequences including test data that make the sequences executable. The core procedure of "Model Learning" was presented in the previous section. Thus one can (a) use the "next refinement" of the Event-B model that contains more information; or (b) one can "provide a counterexample" by manually or automatically providing sequences that are feasible in the Event-B model, but are not in the cover automaton or vice-versa; these counterexamples are used by the learning procedure to make the automaton approximating the Event-B model more precise; or (c) one can increase the bound ℓ and implicitly feed the learning engine with longer sequences which again will increase the precision of the finite state approximation. At any point in the time, one can use the generated cover automaton to generate tests that exercise different sequences through the Event-B model. There are many existing methods for test generation from finite state models. In our case, we use internal information from the learning procedure, which maintains a so-called "observation table" which keeps track of the learned feasible sequences. Sets of feasible sequences in this table will provide the desired test suite. Note that during the feasibility check of the sequences in Event-B, test data are also generated. This is implemented by using a constraint-solver for Event-B [131] and is one of the most time consuming part of the algorithm. The test suite obtained satisfy strong criteria for conformance testing (usually required in the embedded system domain) and may be large. If weaker test coverage like state-, transition- or event-coverage are desired, optimization algorithms[17] are applied on the test suite (see rightmost loop in the Fig.

[17] We implemented different optimizations as proposed by one of the co-authors in [132] using the jMetal framework (http://jmetal.sourceforge.net) based on genetic algorithms.

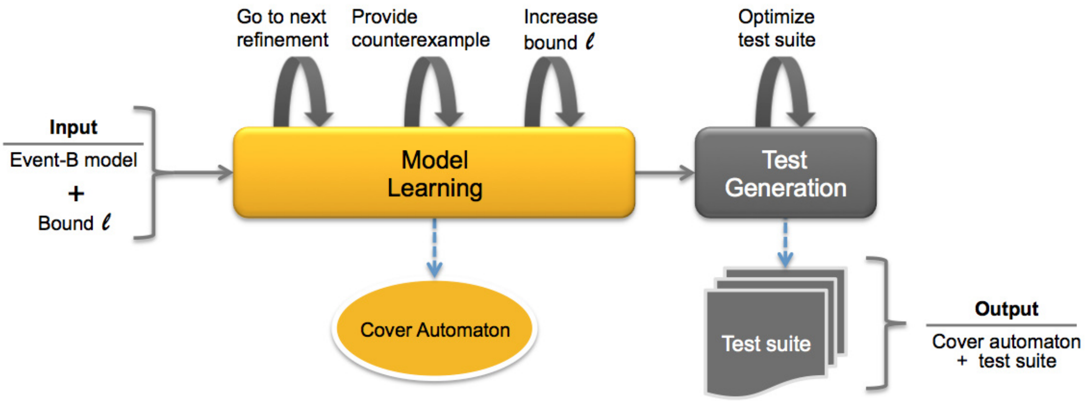


Figure 3.12: Overview of the implemented approach.

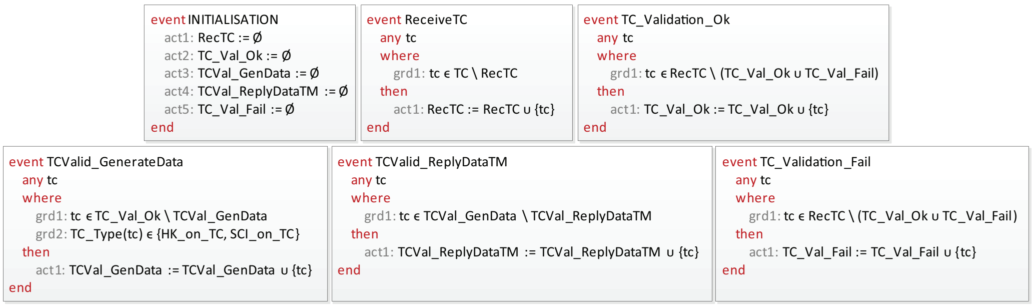
3.12).

An Event-B model for a space craft system. BepiColombo[18] mission is one of the case study used by SSF. The paper [130] provides a rigorous Event-B model description, which we follow next with the scope of introducing the modeling in Event-B.

Briefly, the BepiColombo mission consists of two orbiters. The Mercury Planetary Orbiter is responsible for carrying remote sensing and radioscience instrumentation and an important part of it is the core, which, together with the mission-critical software that controls the whole system, is responsible for controlling the power of devices and their operation states and to handle TeleComand (*TC*) and TeleMessage (*TM*) communications, as follows: (1) A TeleCommand (*TC*) is received by the core from Earth; (2) The core software checks its syntax; (3) After passing the syntactic validation, a semantic checking is carried out on the received *TC*; (4) If the *TC* is valid, a control TeleMessage (*TM*) is generated and sent to Earth, if needed; (5) Some particular *TC*s require more than one data *TM* to be sent back to Earth.

BepiColombo is modeled in Event-B using several levels of refinements (combined with atomic and model decompositions which we do not address here). The main goal of the system is specified at a very abstract level, with a machine M_0. The system specification is concretized through three further refinement levels, M_1, M_2 and M_3. Fig. 3.13 presents the five events of M_0, plus a special event called 'Initialisation'. We see that each event has a local parameter (in this case 'tc'), a guard that decides whether the event is executed or not (for instance the guard of ReceiveTC is $tc \in TC \setminus RecTC$ that check that the input parameter tc is

[18] http://en.wikipedia.org/wiki/BepiColombo



```
event INITIALISATION
   act1: RecTC := ∅
   act2: TC_Val_Ok := ∅
   act3: TCVal_GenData := ∅
   act4: TCVal_ReplyDataTM := ∅
   act5: TC_Val_Fail := ∅
end
```

```
event ReceiveTC
   any tc
   where
      grd1: tc ∊ TC \ RecTC
   then
      act1: RecTC := RecTC ∪ {tc}
end
```

```
event TC_Validation_Ok
   any tc
   where
      grd1: tc ∊ RecTC \ (TC_Val_Ok ∪ TC_Val_Fail)
   then
      act1: TC_Val_Ok := TC_Val_Ok ∪ {tc}
end
```

```
event TCValid_GenerateData
   any tc
   where
      grd1: tc ∊ TC_Val_Ok \ TCVal_GenData
      grd2: TC_Type(tc) ∊ {HK_on_TC, SCI_on_TC}
   then
      act1: TCVal_GenData := TCVal_GenData ∪ {tc}
end
```

```
event TCValid_ReplyDataTM
   any tc
   where
      grd1: tc ∊ TCVal_GenData \ TCVal_ReplyDataTM
   then
      act1: TCVal_ReplyDataTM := TCVal_ReplyDataTM ∪ {tc}
end
```

```
event TC_Validation_Fail
   any tc
   where
      grd1: tc ∊ RecTC \ (TC_Val_Ok ∪ TC_Val_Fail)
   then
      act1: TC_Val_Fail := TC_Val_Fail ∪ {tc}
end
```

Figure 3.13: The events of the abstract machine M_0 in BepiColombo Event-B model [130]

a telecommand (TC) that was not received yet (RecTC)), and an action code (for ReceiveTC, we add the received telecommand tc to the set of already received telecommands by $RecTC := RecTC \cup \{tc\}$). Note that there are global variables like $RecTC$ of type Set, that are initialized in the event 'Initialisation'. Once the 'Initialisation' event is executed, the modeled system moves from one state to another by choosing one event with its guard true and executing its action code. Note that the current state of the Event-B model is not explicit, but given by the value of its global variables. Thus a finite automaton approximating the behavior of the Event-B model would be very helpful in better understanding the control flow of the system.

Learning and test generation approach. The concept of state is at the heart of testing and many test generation techniques from (extended) finite state machines exist. Since Event-B states are not given explicitly, in order to apply the aforementioned techniques, an explicit state model has to be constructed first. Our tool addresses this problem, keeping under control the state space explosion by implementing an incremental model-learning algorithm to iteratively construct a approximation of a subset of a state space.

Given the BepiColombo Event-B model and an upper bound ℓ, we will incrementally construct finite cover automata that will eventually cover all executable event sequences of length less than or equal to ℓ. Figure 3.14 (plotted by our tool) illustrates the cover automaton for the first machine M_0 and $\ell = 4$, which is minimal by construction, having the initial state marked with q_0, transitions labeled with event names and final states marked with a double circle. Starting from the state q_0, the enabled event sequences can be identified by following the transitions with the purpose of reaching the automaton final states, representing a subset of the communication scenarios the space craft system may encounter.

A conformance test suite heavily exercising the system would consist of 17

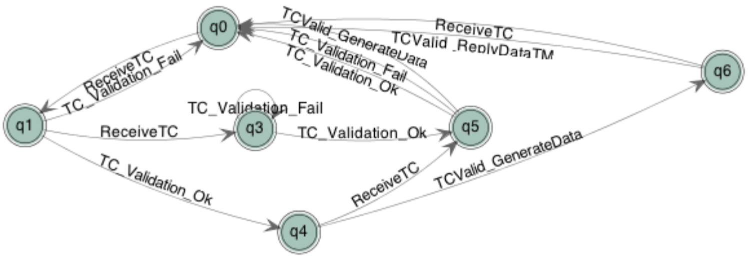


Figure 3.14: The generated cover automaton for M_0 and $\ell = 4$

test cases. However, a test suite covering each event once can consist of 2 tests (of length up to 4): (a) *ReceiveTC(tc1), TC_Validation_Ok(tc1), TCValid_GenerateData(tc1), TCValid_ReplyDataTM(tc1)* and (b) *ReceiveTC(tc2), TC_Validation_Fail(tc2)*.

Also, a list of infeasible paths (sequences of events for which test data was not found, being disjoint form the list of timeout paths) is maintained during the model-learning process and displayed on the same wizard page, e.g. the invalid communication scenario: *INITIALIZATION, ReceiveTC, TC_Validation_Ok, TCValid_ReplyDataTM*.

3.5.2 Tool architecture

Architecture. The tool is a Rodin plugin implemented in Java (over 5,500 LOC), that can be called on any Event-B model with several levels of refinements. The architecture of the tool is presented in Fig. 3.15, where the main classes of the modules are captured (cf. Fig. 3.12). It implements adaptations of Angluin's learning algorithms for cover automata, test generation and optimization algorithms and is integrated with Rodin, ProB and jMetal plugins. A wizard-based UI guides the user execution.

3.5.3 Tool presentation with screenshots

Below we provide some more information on the tool, including installation instructions and screenshots.

Installation

The MBT for Event-B plug-in (also referred to as MBT plugin) relies on Rodin release 2.0 or newer and ProB plugin 2.2 or newer. The following steps guide you through the installation process:

1. Download the latest Rodin release for your platform from Sourceforge (`http://sourceforge.net/projects/rodin-b-sharp`)

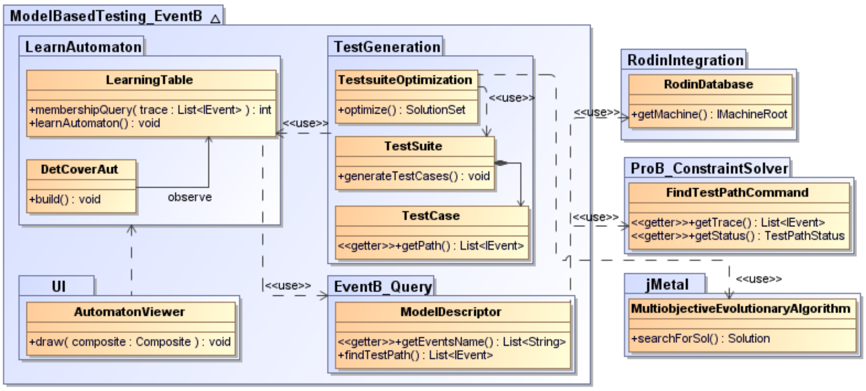

ModelBasedTesting_EventB
LearnAutomaton
LearningTable
+membershipQuery(trace : List<IEvent>) : int
+learnAutomaton() : void
DetCoverAut
+build() : void
observe
TestGeneration
TestsuiteOptimization
+optimize() : SolutionSet
<<use>>
TestSuite
+generateTestCases() : void
TestCase
<<getter>>+getPath() : List<IEvent>
<<use>>
UI
AutomatonViewer
+draw(composite : Composite) : void
<<use>>
EventB_Query
ModelDescriptor
<<getter>>+getEventsName() : List<String>
+findTestPath() : List<IEvent>
RodinIntegration
RodinDatabase
+getMachine() : IMachineRoot
<<use>>
ProB_ConstraintSolver
FindTestPathCommand
<<getter>>+getTrace() : List<IEvent>
<<getter>>+getStatus() : TestPathStatus
<<use>>
jMetal
MultiobjectiveEvolutionaryAlgorithm
+searchForSol() : Solution
<<use>>

Figure 3.15: High level architecture of the tool.

2. Extract the downloaded zip file.

3. Start Rodin from the folder where you extracted the zip file in the previous step.

4. Install the ProB plugin. See ProB installation instructions here: `http://www.stups.uni-duesseldorf.de/prob_updates`

5. Install the MBT for Event-B plugin:

   (a) In the menu choose Help -> Install New Software....

   (b) Click Add....

   (c) As Location enter `http://fmi.upit.ro/mbt_plugin`

   (d) Enter a name e.g. MBT for Event-B Plugin.

   (e) Click Ok.

6. Restart Rodin as suggested.

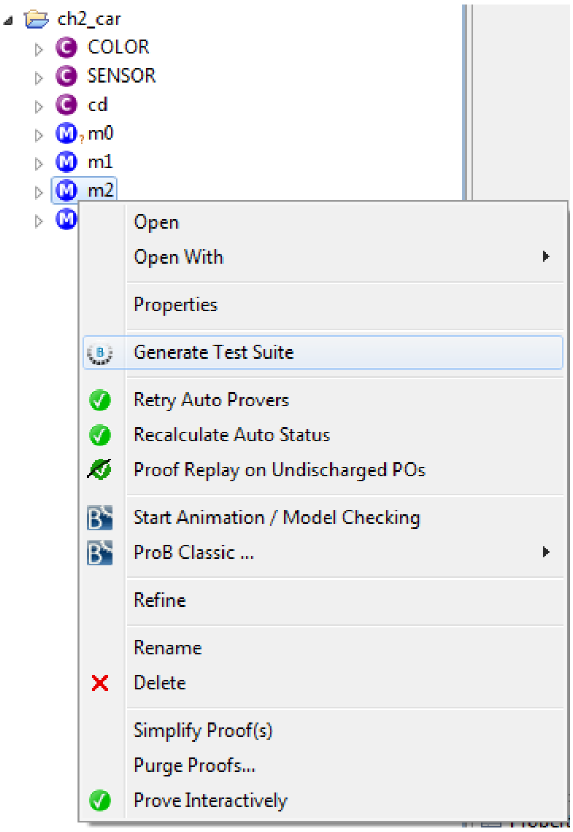
Usage

This subsection should help you to get started with the MBT plugin.

Short theoretical aspects The MBT for Event-B plugin iteratively constructs a subset of the state space of an Event-B model (which is essentially an abstract state machine where the states can be implicitly derived from the values of the model variables), together with an associated test suite, using an incremental model learning which keeps under control the state space explosion. In the following, we will refer to this process as the Learn DFCA algorithm. The state model constructed is based on the concept of deterministic finite cover automaton (DFCA) of a finite set L, which is a deterministic finite automaton which accepts all sequences in L but may also accept sequences that are longer than every sequence in L, with respect to an upper bound ℓ on the length of the considered sequences. Given an Event-B model and an upper bound ℓ, the MBT plugin will incrementally construct finite cover automata that will eventually cover all executable event sequences of length less than or equal to ℓ. Also, a set of test cases associated with the cover automata is evolved during iterations, along with the corresponding test data that makes them executable on the Event-B model. The execution of a test case implies the existence of appropriate test data for the events, i.e. appropriate values for the event parameters in order to ensure that the corresponding guard is true. By definition, a test suite is a collection of test cases. The state model is constructed in a gradual manner, permitting the integration of the Event-B refinement in this process. Whenever the generated cover automaton is found inaccurate, a counterexample path (a collection/sequence of events) must be provided and the Learn DFCA algorithm must be re-executed.

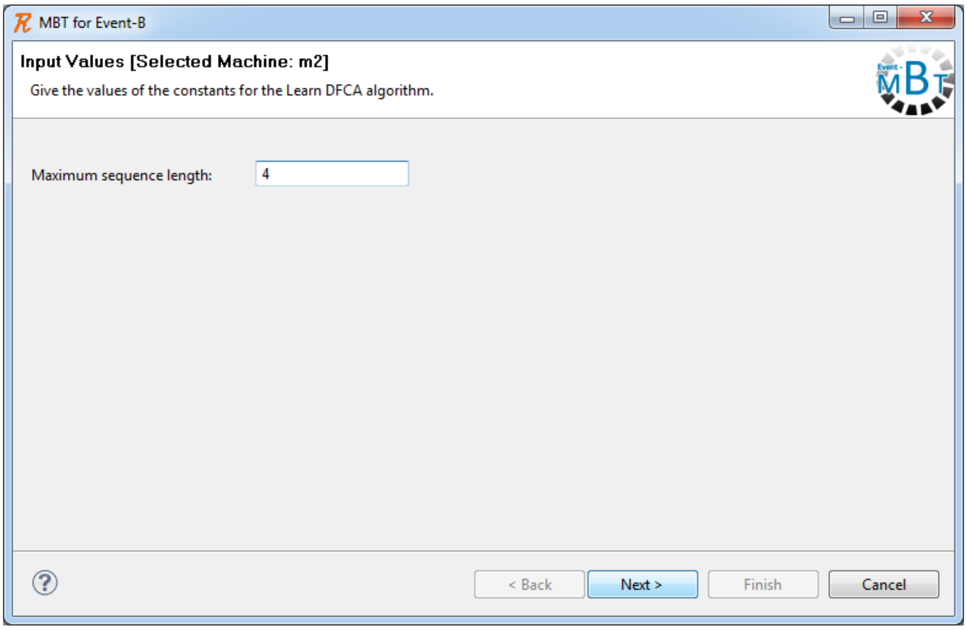
Starting the plugin To start the MBT for Event-B plugin, the "Generate Test Suite" action must be selected from the context menu of an Event-B machine.

The plugin workflow is based on the concept of wizard, guiding the user step by step toward test suite generation, starting with the most abstract machine and ending with the selected one.



Setting the constant value The first step requires the user to specify the value for the maximum sequence length constant ℓ, which is the upper bound on the length of the event sequences accepted by the cover automaton which will be generated.

By clicking the Next button, the Learn DFCA algorithm is executed, computing the deterministic finite cover automaton and the associated test suite along with the corresponding test data.

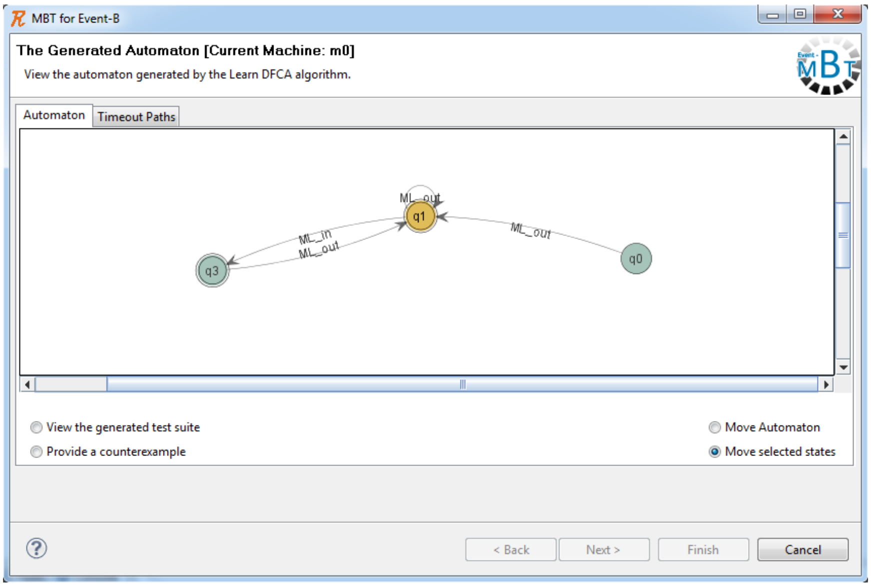

MBT for Event-B

Input Values [Selected Machine: m2]

Give the values of the constants for the Learn DFCA algorithm.

Maximum sequence length: 4

Displaying the generated cover automaton The second wizard page displays the computed cover automaton and (if this is the case) the timeout paths. A timeout path is a sequence of events of length at most ℓ, for which no test data were found to trigger it within a given time bound. In the graphical representation of the automaton, the final states are drawn in double line, whereas non-final states are drawn in single line. The initial state is labeled q_0 and the transitions are labeled with event names.


MBT for Event-B

The Generated Automaton [Current Machine: m0]

View the automaton generated by the Learn DFCA algorithm.

○ View the generated test suite

○ Provide a counterexample

○ Move Automaton

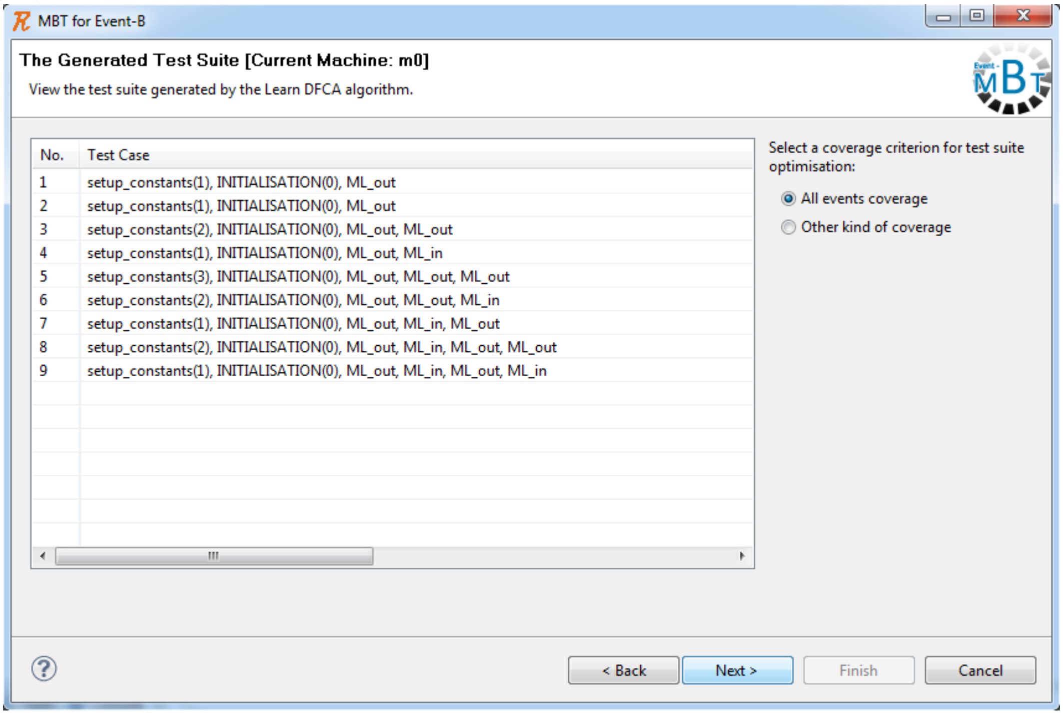
● Move selected states

Based on the user selection, the following options are available:

- moving the generated automaton as a whole
- repositioning selected states of the automaton or highlighting transitions
- displaying the generated test suite
- providing a counterexample path

Displaying the generated test suite By choosing to view the generated test suite, a list of test cases is provided along with the corresponding test data. The MBT for Event-B plugin provides a means for optimizing a test suite according to a chosen coverage criteria, e.g. the all events coverage criterion.

By selecting a coverage criterion and clicking Next, the optimized test suite is displayed.



The Generated Test Suite [Current Machine: m0]

View the test suite generated by the Learn DFCA algorithm.

No.	Test Case
1	setup_constants(1), INITIALISATION(0), ML_out
2	setup_constants(1), INITIALISATION(0), ML_out
3	setup_constants(2), INITIALISATION(0), ML_out, ML_out
4	setup_constants(1), INITIALISATION(0), ML_out, ML_in
5	setup_constants(3), INITIALISATION(0), ML_out, ML_out, ML_out
6	setup_constants(2), INITIALISATION(0), ML_out, ML_out, ML_in
7	setup_constants(1), INITIALISATION(0), ML_out, ML_in, ML_out
8	setup_constants(2), INITIALISATION(0), ML_out, ML_in, ML_out, ML_out
9	setup_constants(1), INITIALISATION(0), ML_out, ML_in, ML_out, ML_in

Select a coverage criterion for test suite optimisation:

● All events coverage

○ Other kind of coverage

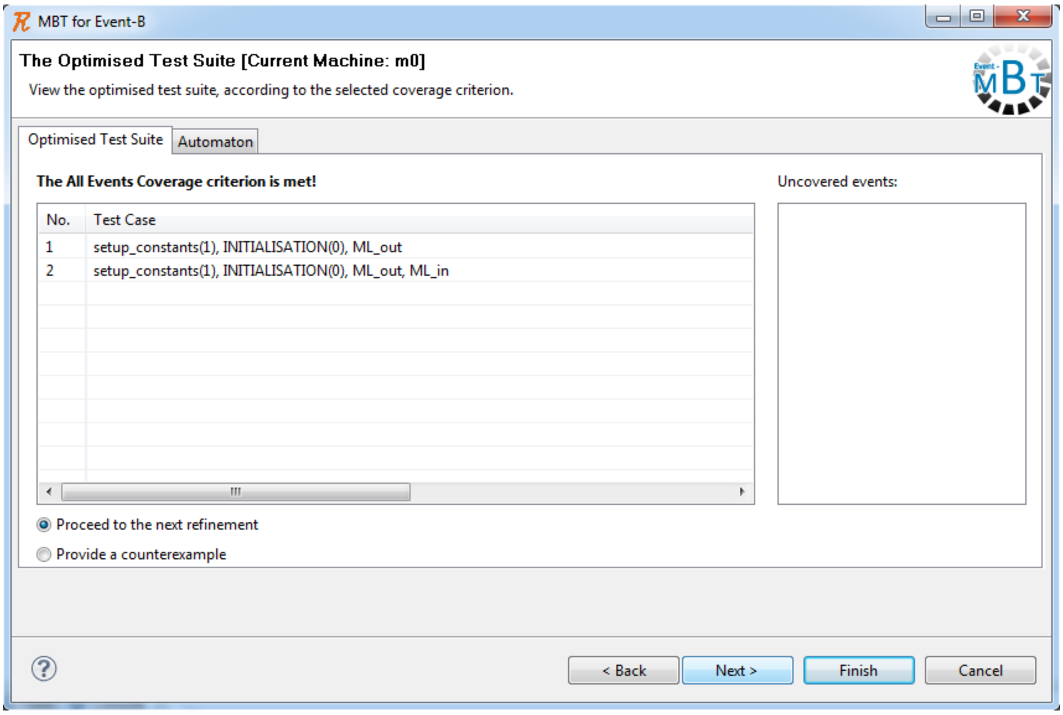
Displaying the optimized test suite It displays a list of test cases along with the corresponding test data, with respect to the selected coverage criterion.

If the all events coverage criterion was selected for test suite optimization and was not satisfied, a list of uncovered events is provided. Based on the user selection, the following options are available:

- proceeding to the next refinement, causing the Learn DFCA algorithm to be executed for a more concrete machine which refines the current one

- providing a counterexample path

By proceeding to the next refinement, it causes for a more concrete Event-B machine (which refines the current one) to be loaded and for the DFCA to be recomputed such that to accept new sequences which include the new events of the loaded machine.



The Optimised Test Suite [Current Machine: m0]

View the optimised test suite, according to the selected coverage criterion.

The All Events Coverage criterion is met!

Uncovered events:

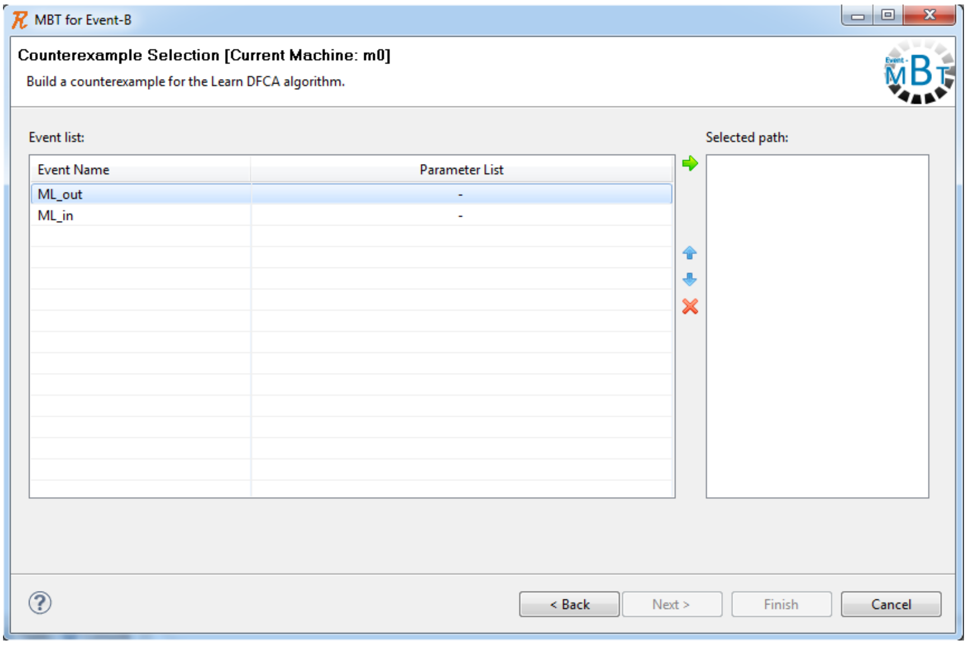
No.	Test Case
1	setup_constants(1), INITIALISATION(0), ML_out
2	setup_constants(1), INITIALISATION(0), ML_out, ML_in

● Proceed to the next refinement

○ Provide a counterexample

Providing a counterexample path A counterexample path is sequence of events of length at most ℓ, which was not accepted by the generated cover automaton. By providing a counterexample, the Learn DFCA algorithm will compute a new DFCA which accepts it, if there exist test data to trigger the given event sequence. The events of the current Event-B machine and their parameters are displayed in the "Event list" table. By double-clicking an event (or selecting it and then pressing the green arrow) it is added to the "Selected path" list. The events from the constructed sequence can be reordered by using the blue arrows or deleted by using the red cross button.

After building a counterexample path and clicking Next, the wizard returns to displaying the computed DFCA.



MBT for Event-B

Counterexample Selection [Current Machine: m0]

Build a counterexample for the Learn DFCA algorithm.

Event list:

Event Name	Parameter List
ML_out	-
ML_in	-

Selected path:

Bibliography

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

[2] J.-R. Abrial. A System Development Process with Event-B and the Rodin Platform. In *Proc. ICFEM'07*. LNCS 4789, pp. 1-3. Springer, 2007.

[3] J.-R. Abrial. *Modeling in Event-B: System and Software Design*. Cambridge University Press, 2010.

[4] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 6:447-466, 2010.

[5] J.-R. Abrial, W. Su, and H. Zhu. Formalizing Hybrid Systems with Event-B. In *Proceedings of the ABZ International Conference (ABZ 2012)*, LNCS, Springer-Verlag, 2012.

[6] R.-J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, 1983.

[7] R.-J. Back, L. Petre, and I. Porres Paltor. Continuous Action Systems as a Model for Hybrid Systems. In *Nordic Journal of Computing*, Vol. 8, No. 1, pp. 2-21, Publishing Association Nordic Journal of Computing, 2001.

[8] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *Proc. of the DIMACS/SYCON Workshop on Hybrid Systems III : Verification and Control*, pp. 232-243. Springer-Verlag, 1996.

[9] J. Berthing, P. Boström, K. Sere, L. Tsiopoulos and J. Vain. Refinement-based development of timed systems. In *Proceedings of the Integrated Formal Methods International Conference (iFM 2012)*, LNCS, Springer-Verlag, 2012.

[10] J. W. Bryans, J. S. Fitzgerald, A. Romanovsky, and A. Roth. Patterns for Modelling Time and Consistency in Business Information Systems. In *Proc. of 15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010*, pp. 22-26, IEEE, 2010.

[11] J. Bryans and W. Wei. Formal Analysis of BPMN Models Using Event-B. In *Proc. of the 15th Int. Workshop on Formal Methods for Industrial Critical Systems, FMICS 2010*. LNCS, Vol. 6371, pp. 33-49, Springer-Verlag, 2010.

[12] D. Cansell, D. Mery, and J. Rehm. Time Constraint Patterns for Event B Development. In *J. Julliand and O. Kouchnarenko, eds, B 2007: Formal Specification and Development in B, 7th Intl. Conf. of B Users*, LNCS 4355, pp. 140-154. Springer-Verlag, 2007.

[13] D. Diaconescu, I. Leustean, L. Petre, K. Sere, and G. Stefanescu. *Refinement-Preserving Translation from Event-B to Register-Voice Interactive Systems*. In *Proc. of the Integrated Formal Methods International Conference (iFM 2012)*, LNCS, Springer-Verlag, 2012.

[14] R. Gramatovici, L. Petre, K. Sere, A. Stefanescu, G. Stefanescu. Synchronization in Timed Interactive Systems and Event-B. TUCS Technical Reports, Shortly available in May 2012, tucs.fi.

[15] A.Iliasov, L. Laibinis, E. Troubitsyna, A. Romanovsky, T. Latvala. Augmenting Event B Modelling with Real-Time Verification. TUCS Technical Reports No. 1006, 2011, tucs.fi. To appear in *Proc. of FormSERA'12*.

[16] M. Kamali, L. Petre, K. Sere, and M. Daneshtalab. Refinement-Based Modeling of 3D NoCs. In *Proc. FSEN'11*, LNCS. Springer-Verlag, 2011.

[17] M. Kamali, L. Petre, K. Sere, and M. Daneshtalab. Formal Modeling and Verifying of Network-On-Chip architectures. Submitted, 2012.

[18] M. Kamali, L. Petre, K. Sere, and M. Daneshtalab. Formal Modeling of Multicast Communication in 3D NoCs. In *Proc. DSD'11*, pp. 634-642. IEEE, 2011.

[19] T. Lecomte. Applying a Formal Method in Industry: A 15-Year Trajectory. In *Proc. of the Formal Methods for Industrial Critical Systems, FMICS 2009*. LNCS, Vol. 5825, pp. 26-34, Springer-Verlag, 2009.

[20] M. Mazzara and A. Bhattacharyya. On Modelling and Analysis of Dynamic Reconfiguration of Dependable Real-Time Systems. In *Proc. of DEPEND 2010 - Third International Conference on Dependability*, pp. 173-181, 2010.

[21] J.-R. Abrial and T. S. Hoang. Using Design Patterns in Formal Methods: An Event-B Approach. In *Proc. ICTAC'08*, LNCS 5160, pp. 1-2. Springer, 2008.

[22] G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, 1986.

[23] R. Back and J. V. Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[24] J.A. Bergstra, A. Ponse, and S.A. Smolka (Eds.). *Handbook of Process Algebra.* Elsevier, 2001

[25] E. Borger and R. Stark. *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer, 2003.

[26] M. Broy. Compositional refinement of interactive systems. *Journal of the ACM*, 44:850-891, 1997.

[27] M. Broy and E.R. Olderog. Trace-oriented models of concurrency. In *Handbook of process algebra* (Eds. J.A. Bergstra et.al.), pp. 101-196. North-Holland, 2001.

[28] M. Broy and G. Stefanescu. The algebra of stream processing functions. *Theoretical Compututer Science*, 258:99-129, 2001.

[29] M. Butler. Decomposition Structures for Event-B. In *Proc. IFM'09*, LNCS 5423, pp. 20-38. Springer 2009.

[30] M. Butler, J. Grundy, T. Løangbacka, R. Ruksenas and J. V. Wright. The Refinement Calculator: Proof Support for Program Refinement. In *Proceedings of Formal Methods Pacific*, pp. 40-61. Springer, 1997.

[31] M. Butler, E. Sekerinski, and K. Sere. An Action System Approach to the Steam Boiler Problem. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control.* LNCS 1165, pp. 129-148. Springer, 1996.

[32] D. Craigen, S. Gerhart, and T.Ralson. Case Study: Paris Metro Signaling System. *Proceedings of IEEE Software*, pp. 32-35, IEEE, 1994.

[33] C. Dragoi and G. Stefanescu. AGAPIA v0.1: A programming language for interactive systems and its typing systems. In: *Proc. FINCO/ETAPS 2007*, ENTCS 203, pp. 69-94. Elsevier, 2008.

[34] C. Dragoi and G. Stefanescu. On compiling structured interactive programs with registers and voices. In *Proc. SOFSEM 2008*, LNCS 4910, pp. 259-270. Springer, 2008.

[35] F. Gadducci and U. Montanari. The tile model. In *Proof, language, and interaction: Essays in honor of Robin Milner*, pp. 133-168. MIT Press, 1999.

[36] D. Goldin, S. Smolka, P. Wegner (Eds.). *Interactive Computation: The New Paradigm.* Springer, 2006.

[37] T. S. Hoang, A. Fürst and J.-R. Abrial. Event-B Patterns and Their Tool Support. In *Proc. SEFM'09*, pp.210-219. IEEE, 2009.

[38] A. Iliasov, E. Troubitsyna, L. Laibinis, and A. Romanovsky. Patterns for Refinement Automation. In *Proc. FMCO 2009*, LNCS, pp. 70-88. Springer-Verlag, 2010.

[39] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala. Supporting Reuse in Event B Development: Modularisation Approach. In *Proc. ABZ 2010*, LNCS 5977, pp. 174-188. Springer, 2010.

[40] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16:872-923, 1994.

[41] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[42] C. Morgan. *Programming from Specifications*. Prentice Hall, 1998.

[43] M. Oliveira, A. Cavalcanti, and J. Woodcock. Arcangel: A Tactic Language for Refinement. *Formal Aspects of Computing Journal*, 15:28-47. Springer, 2003.

[44] A. Pnueli. The Temporal Logic of Programs. In *Proc. FOCS 1977*, pp. 46-57.

[45] A. Popa, A. Sofronia and G. Stefanescu. High-level structured interactive programs with registers and voices. *Journal of Universal Computer Science*, 13:1722-1754, 2007.

[46] M. R. Sarshogh and M. Butler, Specification and refinement of discrete timing properties in Event-B. In *ECEASST*, Vol 46, http://journal.ub.tu-berlin.de/eceasst/article/view/701, 2011.

[47] V.R. Pratt. The duality of time and information. In *Proc. CONCUR'92*, LNCS 630. Springer 1992.

[48] RODIN Project: Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project, online at http://rodin.cs.ncl.ac.uk/.

[49] W.P. de Roever et.al. *Concurrency verification: Introduction to compositional and noncompositional methods*. Cambridge University Press, 2001.

[50] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[51] A. Sofronia, A. Popa, and G. Stefanescu. Undecidability Results for Finite Interactive Systems. *Romanian Journal of Infromation Sciences and technologies*, 12:265-279, 2009. Also: Arxiv, CoRR abs/1001.0143, 2010.

[52] M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 1992.

[53] G. Stefanescu. *Network algebra*. Springer, 2000.

[54] G. Stefanescu. Algebra of networks: modeling simple networks as well as complex interactive systems. In *Proof and System-Reliability, Proc. Marktoberdorf Summer School 2001*, pp. 49-78. Kluwer, 2002.

[55] G. Stefanescu. Interactive systems with registers and voices. Draft, School of Computing, National University of Singapore, July 2004.

[56] G. Stefanescu. Interactive systems with registers and voices. *Fundamenta Informaticae*, 73:285-306, 2006.

[57] G. Stefanescu. Towards a Floyd logic for interactive rv-systems. In: *Proc. 2nd IEEE Conference on Intelligent Computer Communication and Processing* (Ed. A.I. Letia). Technical University of Cluj-Napoca, September 1-2, 2006, 169-178.

[58] URL http://www.petrinets.info/

[59] URL RODIN tool platform, http://www.event-b.org/platform.html.

[60] W. Wadge and E.A. Ashcroft. *Lucid, the dataflow programming language.* Academic Press, 1985.

[61] P. Wegner. Interactive foundations of computing. *Theoretical Computer Science* 192:315-351, 1998.

[62] G. Winskel. An introduction to event structures. In *Proc. REX Workshop*, LNCS 354, pp. 364-397. Springer, 1988.

[63] M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. In *Formal Methods in Systems Design*, Vol. 13, pp. 5-35, Kluwer Academic Publishers, 1998.

[64] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. In *ACM Computing Surveys*, Vol. 41, Issue 4, pp. 1-36, 2009.

[65] Denisa Diaconescu, Ioana Leustean, Luigia Petre, Kaisa Sere, and Gheorghe Stefanescu. Refinement preserving translation from Event-B to register-voice interactive systems. Technical Report no. 1028, TUCS (Turku Center for Computer Science), 51 pp., December 2011.

[66] Denisa Diaconescu, Ioana Leustean, Luigia Petre, Kaisa Sere, and Gheorghe Stefanescu. Refinement preserving translation from Event-B to register-voice interactive systems. To appear in Proc. of Integrated Formal Methods (iFM'12), LNCS, Springer. 2012.

[67] Denisa Diaconescu, Ioana Leustean, Luigia Petre, Kaisa Sere, and Gheorghe Stefanescu. Refinement of Structured Interactive Systems. Submitted to Formal Methods (FM'12) conference, Springer. 2012.

[68] Radu Gramatovici, Luigia Petre, Kaisa Sere, Alin Stefanescu si Gheorghe Stefanescu. Timed interactive systems and their impact on Event-B modelling. To be submitted to TIME'12 conference. 2012.

[69] Alin Stefanescu, Sebastian Wieczorek, and Matthias Schur. Message choreography modeling – a domain-specific language for consistent enterprise service integration. Conditionally accepted at Software and Systems Modeling (SoSyM) journal, 2012.

[70] Ionut Dinca, Florentin Ipate, Laurentiu Mierla, and Alin Stefanescu. Learn and test for Event-B – a Rodin plugin. To appear in ABZ'12, LNCS. Springer, 2012.

[71] D. Harel and M. Politi. *Modeling reactive systems with statecharts: the STATEMATE approach*. McGraw-Hill, 1998.

[72] Samar Mouchawrab, Lionel C. Briand, Yvan Labiche, and Massimiliano Di Penta. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *IEEE Trans. on Software Engineering*, 37(2):161–187, 2010.

[73] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. on Software Engineering*, 36(4):474–494, 2010.

[74] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[75] Cezar Câmpeanu, Nicolae Sântean, and Sheng Yu. Minimal cover-automata for finite languages. *Theoret. Comput. Sci.*, 267(1–2):3–16, 2001.

[76] Cezar Câmpeanu, Andrei Paun, and Jason R. Smith. Incremental construction of minimal deterministic finite cover automata. *Theoret. Comput. Sci.*, 363(2):135–148, 2006.

[77] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.

[78] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. on Software Engineering*, 2012. In Press.

[79] Ionut Dinca, Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose. Test data generation for Event-B models using genetic algorithms. In *Proc. of 2nd International Conference on Software Engineering and Computer Systems (ICSECS'11)*, volume 181 of *CCIS*, pages 76–90. Springer, 2011.

[80] Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. Applying model checking to generate model-based integration tests from choreography models. In *Proc. TESTCOM'09*, volume 5826 of *LNCS*, pages 179–194. Springer, 2009.

[81] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Ed.)*. Addison-Wesley, 2006.

[82] M. P. Vasilevski. Failure diagnosis of automata. *Kibernetika*, 4:98–108, 1973.

[83] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. on Software Engineering*, 4(3):178–187, 1978.

[84] Angelo Gargantini and Elvinia Riccobene. ASM-based testing: Coverage criteria and automatic test sequence. *Journal of Universal Computer Science*, 7(11):1050–1067, 2001.

[85] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the correspondence between conformance testing and regular inference. In *Proc. of FASE'05*, volume 3442 of *LNCS*, pages 175–189. Springer, 2005.

[86] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Proc. of TACAS'02*, volume 2280 of *LNCS*, pages 357–370. Springer, 2002.

[87] Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In *Proc. of FASE'02*, volume 2306 of *LNCS*, pages 80–95. Springer, 2002.

[88] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *Proc. of CAV'03*, volume 2725 of *LNCS*, pages 315–327. Springer, 2003.

[89] Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22(1&2):77–115, 2008.

[90] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Proc. of WCRE'07*, pages 209–218. IEEE Computer Society, 2007.

[91] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *Proc. of ISSTA'02*, pages 112–122. ACM, 2002.

[92] Jens Bendisposto and Michael Leuschel. Automatic flow analysis for Event-B. In *Proc. of FASE'11*, volume 6603 of *LNCS*, pages 50–64. Springer, 2011.

[93] Qaisar Malik, Johan Lilius, and Linas Laibinis. Model-based testing using scenarios and Event-B refinements. In *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *LNCS*, pages 177–195. Springer, 2009.

[94] Jean-Raymond Abrial. *Modeling in Event-B – System and Software Engineering*. Cambridge University Press, 2010.

[95] Renato Silva, Carine Pascal, Thai Son Hoang, and Michael Butler. Decomposition tool for Event-B. *Softw., Pract. Exper.*, 41(2):199–208, 2011. Plug-in webpage: `http://wiki.event-b.org/index.php/Event_Model_Decomposition`.

[96] Thai Son Hoang, Alexei Iliasov, Renato Silva, and Wei Wei. A survey on Event-B decomposition. *ECEASST*, 46:1–15, 2011.

[97] Michael Butler. Decomposition structures for Event-B. In *Proc. of Integrated Formal Methods (iFM'09)*, volume 5423 of *LNCS*, pages 20–38. Springer, 2009.

[98] Renato Silva and Michael Butler. Shared event composition/decomposition in Event-B. In *Proc. of FMCO'10*, volume 6957 of *LNCS*, pages 122–141. Springer, 2010.

[99] Jean-Raymond Abrial. Event model decomposition. Technical Report 626, ETH Zurich, May 2009.

[100] Thai Son Hoang and Jean-Raymond Abrial. Event-B decomposition for parallel programs. In *Proc. of ASM'10*, volume 5977 of *LNCS*, pages 319–333. Springer, 2010.

[101] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010. Tool available online at: `http://sourceforge.net/projects/rodin-b-sharp`.

[102] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines – A survey. *Proc. of the IEEE*, 84(8):1090–1123, 1996.

[103] Florentin Ipate. Bounded sequence testing from deterministic finite state machines. *Theoret. Comput. Sci.*, 411(16–18):1770–1784, 2010.

[104] Florentin Ipate, Ionut Dinca, and Alin Stefanescu. Model learning and test generation using cover automata. Submitted to IEEE Trans. on Software Engineering, 2012.

[105] Ionut Dinca, Florentin Ipate, and Alin Stefanescu. Learn and test for Event-B – a Rodin plugin. In *Proc. of ABZ'12*, LNCS. Springer, 2012. To appear. Plug-in webpage: `http://wiki.event-b.org/index.php/MBT_plugin`.

[106] Florentin Ipate. Learning finite cover automata from queries. *Journal of Computer and System Sciences*, 78:221–244, 2012.

[107] Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the L^* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

[108] P.S. Thiagarajan. A trace consistent subset of PTL. In *Proc. of CONCUR'95*, volume 962 of *LNCS*, pages 438–452. Springer, 1995.

[109] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis, Alexander Romanovsky, Kimmo Varpaaniemi, Dubravka Ilic, and Timo Latvala. Supporting reuse in Event B development: Modularisation approach. In *Proc. of ASM'10*, volume 5977 of *LNCS*, pages 174–188. Springer, 2010. Plug-in webpage: `http://wiki.event-b.org/index.php/Modularisation_Plug-in`.

[110] Michael Poppleton. The composition of Event-B models. In *Proc. of ABZ'08*, volume 5238 of *LNCS*, pages 209–222. Springer, 2008. Plug-in webpage: `http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B`.

[111] Galina Jirásková and Tomás Masopust. State complexity of projected languages. In *Proc. of DCFS'11*, volume 6808 of *LNCS*, pages 198–211. Springer, 2011.

[112] Jacques Julliand, Nicolas Stouls, Pierre-Christophe Bué, and Pierre-Alain Masson. Syntactic abstraction of B models to generate tests. In *Proc. of TAP'10*, volume 6143 of *LNCS*, pages 151–166. Springer, 2010.

[113] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning and integration of parameterized components through testing. In *Proc. of TestCom/FATES'07*, volume 4581 of *LNCS*, pages 319–334. Springer, 2007.

[114] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, and Martin Leucker. Learning communicating automata from MSCs. *IEEE Trans. Software Eng.*, 36(3):390–408, 2010.

[115] Therese Bohlin, Bengt Jonsson, and Siavash Soleimanifard. Inferring compact models of communication protocol entities. In *Proc. of ISoLA'10, part 1*, volume 6415 of *LNCS*, pages 658–672. Springer, 2010.

[116] Sebastian Wieczorek, Alin Stefanescu, and Matthias Schur. Message choreography modeling – a domain-specific language for consistent enterprise service integration. Submitted to Software and Systems Modeling (SoSyM) journal, 2011.

[117] H. Agrawal. Efficient coverage testing using global dominator graphs. In *Proceedings of the 1999 Workshop on Program analysis for software tools and engineering*, pages 11–20, 1999.

[118] J. Black, E. Melachrinoudis, and D. Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on*

Software Engineering (ICSE 2004), pages 106–115, Edinburgh, Scotland, United Kingdom, 2004.

[119] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3), 1979.

[120] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 2001.

[121] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 849–858. Springer LNCS No. 1917., 2000.

[122] J.J. Durillo, A.J. Nebro, and E. Alba. The jMetal framework for multi-objective optimization: Design and architecture. In *CEC 2010*, pages 4138–4325, Barcelona, Spain, July 2010.

[123] M. Harman. Making the case for MORTO: Multi Objective Regression Test Optimization. In *Proceedings of the 1st International Workshop on Regression Testing (Regression 2011)*, Berlin, Germany, 2011.

[124] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

[125] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and T. Latvala. Developing mode-rich satellite software by refinement in Event B. In *15th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2010)*, CCIS, Antwerp, Belgium, 2010.

[126] M. Marre and A. Bertolino. Using spanning set for coverage testing. *IEEE Transactions on Software Engineering*, 29(11):974–984, 2003.

[127] S. Yoo and M. Harman. Pareto efficient multi-objective test-case selection. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 140–150. ACM Press, 2007.

[128] S. Yoo and M. Harman. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *Journal of Systems and Software*, 83(4):689–701, 2010.

[129] E. Zitzler, M. Laumanss, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. *Tech. Rep.*, 103, 2001.

[130] Asieh Salehi Fathabadi, Abdolbaghi Rezazadeh, and Michael Butler. Applying atomicity and model decomposition to a space craft system in Event-B. In *Proc. of 3rd NASA Formal Methods (NFM'11)*, volume 6617 of *LNCS*, pages 328–342. Springer, 2011. See `http://eprints.ecs.soton.ac.uk/22048/`.

[131] Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, 2008. Tool webpage: `http://www.stups.uni-duesseldorf.de/ProB`.

[132] Ionut Dinca. Multi-objective test suite optimization for Event-B models. In *Proc. of Int. Conf. on Informatics Engineering and Information Science (ICIEIS'11)*, volume 251 of *CCIS*, pages 551–565. Springer, 2011.

[133] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. ICSE'11*, to appear.

[134] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools, Object Technology*. Addison-Wesley, 1999.

[135] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[136] Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. Estimating the feasibility of transition paths in extended finite state machines. *Autom. Softw. Eng.*, 17(1):33–56, 2010.

[137] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *FOGA*, pages 69–93, 1990.

[138] Klaus Meffert et al. JGAP - Java Genetic Algorithms and Genetic Programming Package. *http://jgap.sf.net*. Last visited, March 2011.

[139] Raluca Lefticaru and Florentin Ipate. Functional search-based testing from state machines. In *Proc. ICST'08*, pages 525–528. IEEE Computer Society, 2008.

[140] Phil McMinn. Search-based software test data generation: A survey. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[141] Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1013–1020. ACM, 2005.

[142] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (3rd ed.)*. Springer-Verlag, London, UK, 1996.

[143] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.

[144] Manoranjan Satpathy, Michael Butler, Michael Leuschel, and S. Ramesh. Automatic testing from formal specifications. In *Proc. TAP'07*, volume 4454 of *LNCS*, pages 95–113. Springer, 2007.

[145] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition, 2007.

[146] Alin Stefanescu, Florentin Ipate, Raluca Lefticaru, and Cristina Tudose. Towards search-based testing for Event-B models. In *Proc. of 4th Workshop on Search-Based Software Testing (SBST'11)* from ICSTW'11. pp.194-197. IEEE Computer Society, 2011.

[147] Nigel James Tracey. *A Search-based Automated Test-Data Generation Framework for Safety-Critical Software*. PhD thesis, University of York, 2000.

[148] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[149] Thaise Yano, Eliane Martins, and Fabiano L. de Sousa. Generating feasible test paths from an executable model using a multi-objective approach. In *Proc. ICSTW'10*, pages 236–239. IEEE Computer Society, 2010.