MASTER IN COMPUTER SCIENCE

Automated verification of state-based specifications against scenarios: a step towards relating inter-object to intra-object specifications

Bontemps, Yves

Award date:
2001

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Automated Verification of State-Based Specifications against Scenarios

A Step Towards Relating Inter-Object to Intra-Object Specifications

Yves BONTEMPS

Mémoire réalisé en vue de l'obtention du titre de Maître en Informatique

Année Académique 2000-2001

Abstract

While designing a system, it is critical to ensure that its behavioral specification is correct with respect to its requirements. These requirements are often described as a set of scenarios instantiating use-cases. Formal and automated verification of this correctness is desirable in this context.

To formally describe scenarios, we use LSCs [Harel 98a], an extension of Message Sequence Charts (MSCs) allowing the expression of both safety and liveness conditions. An algorithm to automatically translate LSCs into temporal logic is presented. The obtained formulae can then be used by a model-checker to prove the correctness of the specification. Hence, a total automation of the verification process is obtained.

To achieve a better efficiency in verification, we then refine the translation, by splitting the formula into several smaller formulae. Again, we exhibit an algorithmic solution to support these methods.

Keywords : Verification, Live Sequence Charts, Message Sequence Charts, Model-Checking, pivot, scenario, Requirements Engineering, distributed and reactive systems.

Résumé

Lors de la conception d'un système informatique, il est crucial de s'assurer que la spécification comportementale du système respecte les exigences exprimées *a priori* à son propos. Ces exigences prennent le plus souvent la forme d'un ensemble de scénarios.

Dans ce mémoire, les scénarios seront décrits sous la forme de LSCs [Harel 98a], une extension des Message Sequence Charts (MSCs), permettant l'expression à la fois de conditions de *vivacité* et de *sécurité*. Nous présentons un algorithme effectuant la traduction d'un scénario en LSCs vers une formule de la logique temporelle. La formule obtenue peut alors être utilisée par un model-checker pour démontrer la correction de la spécification. Par conséquent, le processus de vérification peut être entièrement automatisé.

Afin d'obtenir une plus grande efficacité lors de la vérification, nous affinons la traduction en divisant la formule en plusieurs formules plus courtes. A nouveau, nous proposons une solution algorithmique en support à ces méthodes d'optimisation.

Mots-clefs : Vérification, Live Sequence Charts, Message Sequence Charts, Model-Checking, pivot, scénario, ingéniérie des besoins, systèmes distribués et réactifs.

Automated Verification of State-Based Specifications against Scenarios
A Step Towards Relating Inter-Object to Intra-Object Specifications.

ERRATA

1. L'imprimeur a interverti les pages suivantes lors de la reliure :

   - Pages 9 et 10.
   - Pages 29 et 30.

2. De plus, à la page 77, (section 4.3.2 : Non Automaton-based technique), à la dernière ligne, il faut lire :

   > Consequently, it could be possible to find pivots in a trace set in time exponential in the size **of the chart**, (…).

Je vous prie d'accepter mes excuses pour les désagréments causés par ces erreurs.

Yves Bontemps.

Contents

List of Figures

List of Tables

List of Algorithms

Foreword

This MSc thesis summarizes the work that I have done during more than two years.

It all started in July 1999, when I had a first contact with the vast field of Requirements Engineering with Scenarios as a summer student, at the Institut d'Informatique of the University of Namur, Belgium. I had the opportunity to collaborate on the implementation of the CREWS distributed animator, thanks to Prof. Pierre-Yves Schobbens and Mr. Patrick Heymans.

They both became my advisors. They have all the qualities that a Master student could want from his promotors. They have always been available to answer my questions and to give me (a lot of) enlightening comments about my work. Normally, [Higham 98], I should not thank people for doing their job. However, they did it so well, with so much devotion, that they fully deserve these acknowledgements. Working with them has been a real pleasure.

In September 1999, I looked for a Master thesis topic, preferably related to the work already accomplished, since a reusability approach is always desirable, from a student's point of view [1]. I read a dozen of articles about Requirements Engineering. Among them, [Harel 98a] presented the role of automatable links between functional requirements and behavioral specification in a way that convinced me that this field was interesting enough to keep me as busy as a bee during two years. As a matter of fact, my interest in this problem grew so much that I decided to spend the coming years addressing the several problems raised by David Harel's round-trip approach to requirements engineering.

Prof. David Harel gave me the opportunity to perform an internship at the Applied Mathematics and Computer Science Department of the Weizmann Institute of Science in Rehovot, Israel. There, I collaborated with Hillel Kugler on the subject presented here. This first international experience has been a great personal enrichment. Thank you, David and Hillel, it has been great to work with you and I hope that we will continue our cooperation in the future.

A considerable part of my work consisted of a case study. For the advice they gave me about it, I would like to thank Prof. Jean Ramaekers and Olivier Bonaventure.

For helping me to justify the undecidability of pivots, I would like to thank Prof. Yves Deville.

During the writing of this thesis, I became aware that it was not easy at all to communicate ideas, even (or especially) when they are expressed as formal theories. I tried as much as possible to follow the advices given in [Higham 98], in order to facilitate the difficult task of my reader. I am quite confident in the result, as the few people who proofread my work seemed to understand the intuition at the center of the presented results. For making an effort to read (even a part of) this dissertation, I would like to thank Simon Brohez and Alain Dallons.

In order to write this report, I used the LaTeX macro package [Lamport 94a] for TeX [Knuth 84]. This old-fashioned system is still, from my point of view, the most efficient tool to produce a professional-looking document with little or no particular effort from the writer. I used several packages, among which `fancyheadings`, `algorithms` and `a4`. The bibliography has been typeset using a modified version of the `these` bibliography style file. All these packages are available on `CTAN`. I thank Cristel Pelsser for her help in this domain. The numerous figures and schemas have

[1] "Unfortunately", the only thing that I could reuse from my previous work was the experience that I gained. This thesis has thus been written from scratch.

been produced using `xfig` or `jfig`. The UML class diagrams documenting the implementation have been drawn with `ArgoUML`, [ArgoUML 98].

For all the support they always gave me, I would like to thank my whole family: Evelyne, Jami, Maman and Papa. I also thank my grand-parents without whom this report would not have been written.

Finally, I would like to thank my girlfriend, Roxane, for having raised so easily my spirit, every time I needed it.

Yves Bontemps,
Namur, May 2001.

Introduction

From problem analysis to system specification

It is a good practice to specify a system before implementing it. The specification phase aims at describing *what* the system will do, while the implementation expresses precisely *how* it does it.

Problem typology

Three perspectives for problem analysis

Traditionally [Pohl 96], a problem could be analyzed under three perspectives:

- information,
- function,
- and control.

The informational specification deals with the problem of data structuring. It identifies the main components of the problem and their static inter-relationships. In the eighties, entity-relationship approaches were mainly used to perform this analysis [Bodart 94].

The functional specification decomposes the functionality of the system under development into sub-functionalities, until it reaches an *atomic* level (in which the functions are specified in structured English). In the eighties, inter-relationships between functions were emphasized through the use of Data Flow Diagrams.

Finally, the control (or dynamic) approach structures the problem as a *decision making* problem. Historically, it has been mainly based on decision tables or state machines.

Of course, it appeared that these three approaches could not be considered as independent. They were different views of a same problem and, thus, had to be considered as a whole. The importance of the different approaches differ depending on the kind of problem to solve, as emphasized in [Jackson 99].

In order to integrate these three views of the problem, object-oriented approaches [Jacobson 99] have been developed. As an example, UML provides the specifier with a set of notations, called *ClassDiagrams*, to *statically* structure the problem as a set of inter-related objects. Since the specifier gives explicitly the information embedded in every object, as well as the methods applicable on it, it merges the classical *data* and *functional* approaches.

After this static specification phase, the problem's objects have been identified. But the specification process is not over. The question "How do these components behave ?" has to be answered. To do so, for instance, UML provides us with a notation called *StateDiagrams*.

In order to understand why a behavioral specification is needed, think about an engineer building a car engine. In the static approach, he will identify the many pieces used in the construction, such as the cylinders, the clutch or the electronic devices. He will even say how they must fit together. But he will not obtain a running engine until he expresses how the pieces will actually move, breathing life into the system.

Transformational versus reactive problems

By nature, some problems are transformational while others are reactive. Traditional computer science[2] deals with transformational systems. When started, they

[2] In the computability theory's sense.

compute a final result from input parameters, give this result to the user and eventually terminate. Loans computations or compilers are typical transformational systems.

Reactive systems never stop [Manna 95]. They keep an ongoing relationship with their environment. They adapt their internal state and they possibly act on the environment, according to stimuli from the surrounding environment. The use of such systems is growing year after year. Graphical-user interface based software (GUI) and embedded systems, the latter being often implemented as hardware, are typically reactive. Nuclear reactor controllers or cell phones are reactive systems, for instance.

The main difference between a transformational system and a reactive system comes from the fact that the latter can receive stimuli from its environment at any time of its execution. Obviously, the least we ask from a computer system is to properly react to a stimulus. This problem is not trivial, as the specifier has to take into account a high combinatorial number of possible executions.

Intra-object specification

A commonly accepted framework for specifying the behavior of reactive systems are *state machines.* They model the system as a finite set of states, linked by transitions. The modelled component changes its internal state, depending on external stimuli. The specification process is often facilitated by dividing the system into more manageable classes of objects, the behavior of which will be described separately. Thus, the specifier provides one state machine by class of objects. The language of state machines is extended to cope with the communication of components, often by message passing or broadcasting of events. Because this view of the behavioral specification describes every possible execution of every object composing the system, it is called an *intra-object specification.* State machines focus on components, one by one; they provide an *all-stories-for-one-object* description of the system [Harel 00a, Harel 98a].

Taking user requirements into account

One could think that the main issue with these highly complex systems, made of simple components put together, is to correctly implement them. It is not. During the past decade, tools have been developed to generate executable and readable code from a behavioral specification, in a fully-automated fashion.

The most difficult problem when modelling such systems is to ensure that they are compliant with the user requirements.

As stated in [Jackson 99], "the requirement (...) is an explicit description of the behavior and properties that we want the world to have as a result of its interaction with the machine. (...) It captures the purpose for which the machine is to be built and installed."

Detecting any inadequacy in the model, with respect to the requirements, has a very high pay-off. Actually, [Davis 93] stresses two empirically verified facts: early detection of requirements errors is *cost-effective* and these errors are mostly due to *unsuitabilit of the expressed requirements and the user's real needs.*

Of course, the question of how to represent requirements, so that they can be easily formulated, understood, tested and refined into a more precise specification, immediately comes up.

Requirements typology

First of all, it is important to keep in mind that there are different kinds of requirements. Loosely speaking, there are two groups: *functional* requirements and *non-functional* requirements.

The former are often seen as *use-cases* [Jacobson 92], that is typical functionalities that the system under development should provide. Performing a call, browsing the Internet or sending a short message are possible use-cases for a cell phone, for instance.

The second group contains everything the user asks from the system which is not functional, such as response time, portability or technical issues, for example.

Secondly, one should be aware that in the system's development process, requirements are expressed at different decomposition levels [Davis 93]. At every level, the means chosen to achieve the preceding level's requirements become requirements for the current level.

In this thesis, we only focus on the functional requirements.

Expressing requirements with scenarios: an inter-object approach

Scenarios have been used for long to describe user requirements, in an intuitive fashion [Jarke 98, Jacobson 92, CREWS 99]. They present partial executions. From the user's point of view, scenarios are advantageous because of their concreteness. They are simply description of *what* the user will do with the system under construction. Moreover, they are partial, in the sense that they do not target at describing every possible behavior of the whole system. Each scenario represents a fragment of one[3] possible execution, projected on the points of interest. Hence, they contribute in reducing the mental load of the requirements process, by allowing the stakeholder to focus on particular aspects of the system and thus, they greatly improve the readability and the quality of the requirements.

Message Sequence Charts (MSCs) are a popular notation to describe scenarios, in the realm of distributed systems. By an abstraction process, they allow the representation of one or several execution traces, putting forward the interactions between the system components and ignoring their internal details.

In contrast with the state machine specification, the requirements, presented as a set of scenarios, grouped in use-cases, will be called the *inter-object specification* of the system. Every scenario provides a *one-story-for-all-objects* description of the system to be, in the user's point of view. Sometimes, to stress the difference between the two points of view, the intra-object specification is called *implementation* while the inter-object specification is simply referred to as *specification.*

Bridging the gap between requirements and specification

Thus, on the one hand, we have an inter-object specification, which expresses what the system must do, but in an incomplete fashion, and, on the other hand, an intra-object specification which says how the system will behave, in a general, possibly inconsistent, way. The question that arises is thus, naturally:

> **How can we relate an intra-object specification to an inter-objet specification, in order to ensure that the first is compliant with the latter?**

[3] Sometimes, several executions can be expressed in the same scenario.

Obviously, there are two possible directions:

1. from the requirements to the behavioral specification.
2. from the behavioral specification to the requirements.

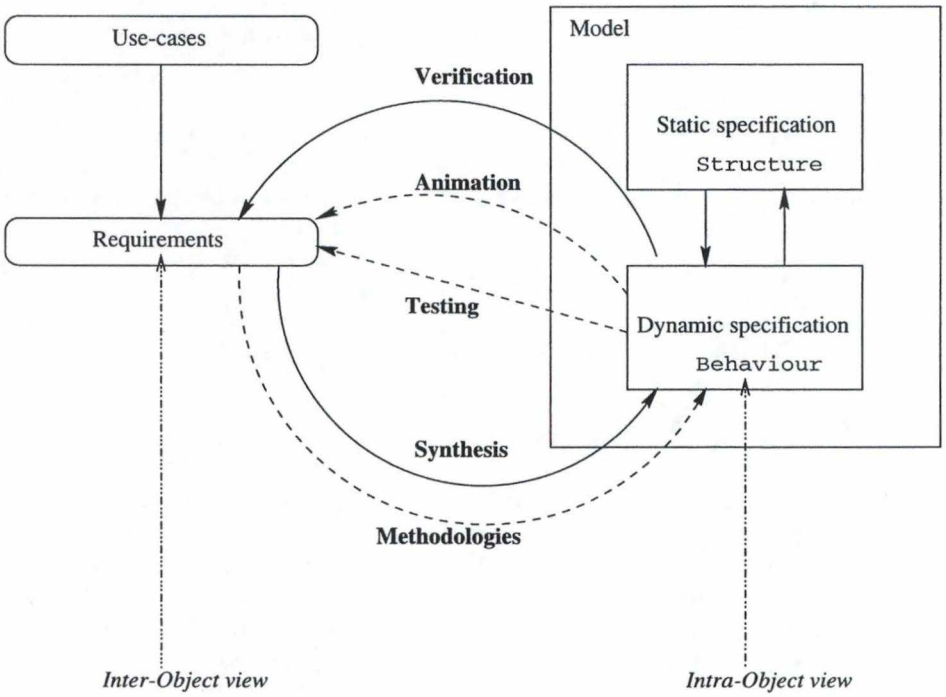


Figure 1: How requirements, static and dynamic specifications fit together.

These two problems are complementary. This round-trip approach to software engineering has been investigated in [Systa 00]. They are illustrated by figure 1[4]. Dashed arrows denote partially automated links between the two kinds of specifications, while solid arrows symbolize fully automated relations. Rounded boxes represent elements that belong to the *problem domain* while plain boxes are parts of the *solution.*

From the requirements to the specification

The first problem is known as *synthesis.* Its goal is to build a specification from the requirements. Traditionally, methodologies [Jacobson 99, Selic 94, Bodart 94], were used to produce the model. Those are based on a long experience in industry and generalize good practices into informal recommendations. Their problem is that they can be wrongly used or not used at all. Their main advantages are the strong experimental background on which they are based and their wide scope. They are not restricted to functional requirements and, actually, can take into account every type of requirements. Consequently, flexibility is their strength and weakness.

Lately, techniques for automatically synthesizing state machines from scenarios have been developed [Biermann 76, Khriss 99, Harel 00b, Whittle 00]. However, much work is still to be done in this field, mainly to ensure a good readability of the produced specifications.

[4]This figure is inspired from [Harel 00a]

From the specification to the requirements

The second problem is known as *validation and verification.* A lot has been done in this field, either to develop tests methodologies, or to involve the non-technical stakeholders in the validation process. Animation, for example, is a widely used means to check specifications correctness but also to refine and discover new scenarios by allowing the user's to *play* with a mock-up of the system to be [Siddiqi 97, Heymans 98]. The main issue with the techniques of testing and animation is that they do not provide an exhaustive testing of the system.

To be sure that a system, in every possible execution, is correct, a verification process is needed. This is equivalent to executing the system during an infinite time. For long, formal proofs of correctness were done manually or in a semi-automated fashion. However, for the past two decades, the techniques of *automated* verification developed widely [Peled 00].

Our work focused on this last point. We investigated how we could formally prove that a specification of a system, given in a state-based language, is correct with respect to its requirements, expressed in an MSC-like language. Our goal is to provide tools performing such proofs in a fully automated fashion.

Master thesis structure

In chapter 1, we will present the visual formalism used to describe scenarios. It is an extension of MSCs, a language standardized by the ITU-T (previously, CCITT) [ITU-T 96], called *live sequence charts (LSC)* [Harel 98a]. It extends MSCs in order to cope with their reduced expressiveness. Mainly, it allows the user to express *mandatory* or *universal* ("the system must *always* behave like that"), *possible* ("this is a *possible* behavior of the system") or *prohibitive* ("the system *may not* behave this way") scenarios.

The subset of the LSC language that we use will be illustrated on the running example of a distributed system achieving mutual exclusion.

Chapter 2 introduces the formalism of state machines. It is used to describe the internal behavior of objects instances. Then, after a short introduction to temporal logics, we will present how the language of LSCs and the language of state-machines can be reduced to a common third language, infinite traces, in which they can be compared. As soon as we will be able to compare requirements and specification, we will precisely define what the concept of *satisfaction* is. Finally, we will show that the verification problem can be reduced to the problem of model-checking of a temporal logic formula, solved by existing automated tools.

Chapters 3 and 4 describe the main and most original part of our work. First, we exhibit an exponential-time algorithm to perform the translation of scenarios into a temporal logic formula, using acyclic non-deterministic finite automata. Secondly, we present an algorithmically supported method to improve the verification process by reducing the size of the formula produced by our first algorithm. The concepts of this method are generalized and the algorithm we propose is polynomial-time.

Chapter 5 presents our implementation of a tool to translate scenarios into temporal logics. This prototype, developed in Java, is editor- and model-checker-independent.

Chapter 6 presents some experimental results of verification.

Chapter 1

Inter-Object Specification

1.1 Live Sequence Charts (LSCs)

1.1.1 Overview

Message sequence charts (MSCs) are a popular mean for specifying scenarios that capture the communication between processes or objects. They are used to describe scenarios in which several objects interact.

They are present in many methodologies and are part of the UML, where they are called sequence diagrams, [Jacobson 99]. There is also a standard for the MSC language, which has appeared as a recommendation of the ITU, [ITU-T 96].

While it is broadly used, this language suffers a lack of expressive power. The semantics of the language is a rather weak ordering of events.

To cope with this problem, Damm and Harel have extended the MSC language into a much richer language, **live sequence charts (LSCs)** [Harel 98a]. The main addition is *liveness* or *universality*, which provides constructs for specifying not only *possible* behavior, but also *mandatory* behavior.

It is thus possible with this language to build scenarios saying that the system *must always* behave like this, or that the system *must at least once* behave like that. This language also enables the construction of *no-stories*, saying that the system *may never* exhibit such a behavior.

To do so, most objects used in the language must be in one out of two exclusive modes. This property is called *temperature* and ranges over two values: hot and cold. Loosely speaking, a hot object represents a *mandatory* requirement while a cold object has a *provisional* meaning. In our work, we will only consider some objects as bi-modal.

LSCs as they are defined in [Harel 98a] are a very rich language allowing conditions, actions, multi-level decomposition and iteration. Our work only focuses on a subset of this language, which we consider as the core of LSCs, since it describes, in a much more powerful way than MSCs would do, the communication behavior of instances.

[Krüger 00] gives an exhaustive and comparative view of the MSCs dialects.

1.1.2 Constructs of the language

Instances and locations

A live sequence chart or, shortly, a *chart*, is made of *instances*. They are represented by vertical lines, along which the time runs, from top to bottom. An instance line consists of several *locations*, each of them linked to one *event*, excepted, perhaps, the first location.

The first location of an instance line is called the *initial* location while the last location is known as the *maximal* location.

An event is either the sending or receiving of a message, to, or from, another instance. An event related with a location occurs *before* the event related with any *lower* location on the same instance line.

This chronological order only applies when considering one instance. If we take two distinct instances, no conclusion may be drawn solely from the graphical representation of their locations.

Considering figure 1.2, one can see that there are three instances, named *Node1,Node2,CS*. Their locations have been labelled by nonnegative integers. In instance *Node2*, for example, location *2* comes after location *1*.

Locations can be *hot* or *cold*. Cold locations are represented by a dashed line segment while hot locations are drawn as a solid line segment.

Labelling a location with the temperature *hot* entails that the chart *must* progress beyond the location, along the subsequent segment of the instance line.

The maximal locations must be cold because when an instance has reached its last location, we cannot oblige it to go any further!

Messages

The ordering between locations in distinct instances can be deduced from the messages. The reception of a message must always occur after its emission. Thanks to this constraint, we are able to order some locations in different instances.

Messages can be either *synchronous* or *asynchronous.*

They are represented by an arrow, going from the location associated with the *sending* event to the location associated with the *receiving* event. Synchronous messages have a solid arrow-head and asynchronous messages, an open-ended one.

If a message is synchronous, it blocks the sender until the reception of the message. Therefore, its sending comes before its receiving, which must precede every event following the sending. Only the sender and receiver of a synchronous message are concerned with this *blocking issue.* Every other instance ignores what happens in these two instances and thus, may proceed along its execution thread.

Referring to figure 1.2, one can see that the messages *request* and *reply* are *asynchronous* while the message *enter* is synchronous.

On figure 1.4, the location on which the *request* is sent comes *after* the location on which the *exit* is received, because *exit* is a synchronous message.

Activation

Charts may be activated by the reception of an *activation message* emanating from the environment. When this event occurs, the main chart applies. As an example, consider the first message in the chart of figure 1.2.

They can also be activated by the so-called *precharts.* These are charts describing an activation protocol. When the instances exhibit the behavior described by this protocol, the main chart applies. Precharts are surrounded by a dashed hexagonal frame. Refer to figure 1.4, for an example of a prechart.

Coregions

Sometimes, it can be useful *not* to order some events along the same instance line. Consider the chart of figure 1.5. We just want to keep Node1 and Node2 from entering into the critical section. The order in which they ask to enter is irrelevant.

The language of LSCs offers us the mechanism of *coregions* to do this. When some locations of the same instance are put into a coregion, they become unordered. Coregions are represented by a dashed vertical line, parallel with the instance line, that runs over the locations belonging to the coregion.

Restricted events

With every chart, we associate a set of events, called the *restricted events.* At least every event appearing in the chart is included in this set. But some other events may also be restricted. Between two events explicitly described in the chart, the events restricted *may not* appear. See figures 1.5 and 1.7.

Temperature of a chart

A hot chart is called a *universal chart.* Every time it is activated, by either an activation message or a prechart, its whole behavior *must* apply. Note that if the chart is never activated, the system will trivially satisfy it. A system ensuring that every chart can be activated is called an *healthy* system.

Universal charts are surrounded by a solid box.

"Every time the call button of a lift is pressed, the lift will stop at this floor" is a scenario suitably described by a universal chart.

For an example, consider the chart of figure 1.2 or figure 1.3.

A cold chart is called an *existential chart.* Among all the possible executions of the system, there must be at least one corresponding to the behavior of this chart. "It is possible that a lift stops at every floor" would be properly described as an existential chart.

Existential charts are drawn in a dashed box.

Figures 1.6 and 1.8 are examples of existential charts.

Universal and existential charts are used to describe *yes-stories*, i.e. desirable properties of the system. They are used to tell that a system *must* or *may* exhibit a behavior.

No-stories

But, sometimes, it can be useful to explicitly forbid some behaviors, i.e. to express *no-stories* or *anti-scenarios.* For example, "the floor doors of a lift *may not* open if the car is not waiting at this floor". With such a mechanism, we are able to keep bad things from happening. These no-stories are represented by *existential charts* headed by a box in which "NOT" is written [1] . The "mutual exclusion" chart is an example of a no-story (figure 1.5).

A no-story is genuinely nothing but an existential chart that *may not* happen. So, in the sequel, we will not take any special care of these charts. One should just remember that every result applying to an existential chart also applies to a no-story, after having been negated.

1.1.3 Abstract syntax

We suppose that there exists a set $Events \subset Prop$, consisting of all the observable events that may appear in the system, where $Prop$ is a set of *propositions.* Of course, we require $Events$ to be finite and we suppose that $Events$ contains $\perp$, meaning that no event occurs.

Definition 1.1 (Basic chart m) *A* basic chart *or, simply, a* chart m *is a tuple* $< dom(m), coreg(m), ev(m), temp(m), Events(m), \rightsquigarrow_m >$, *in which*

- $dom(m)$ *are the locations in* m.
- $coreg(m)$ *are the coregions in* m. *A coregion is a set of locations belonging to the same instance.*
- $ev(m) : dom(m) \nrightarrow Events(m)$ *associates an event with every location except, possibly, the first one of an instance.*
- $temp(m) : dom(m) \rightarrow \{hot, cold\}$ *is the temperature of every location.*
- $Events(m) \subseteq Events \subset Prop$ *are the restricted events in* m.
- $\rightsquigarrow_m \subseteq dom(m) \times dom(m)$ *is a relation on* $dom(m)$ *where* $l_1 \rightsquigarrow_m l_2 \iff l_1$ *is the* send *location of a message and* l_2 *is the* receive *location of this message.*

[1] No-stories in [Harel 98a] are represented using universal charts and hot locations, stating that the condition *must* always be evaluated to true. Because we ignore internal behavior (our LSCs have no access to local variables), we do not have conditions. Therefore, we use an explicit formalism to describe them.

Definition 1.2 (Predecessor/Successor of a location) *For two locations* $l_1, l_2 \in dom(m)$, *we say that* l_1 *is a* predecessor *of* l_2 *or, dually,* l_2 *is a* successor *of* l_1, *and we write it* $l_1 \prec_m l_2$, *if all the following conditions hold:*

1. l_1 *and* l_2 *belong to the same instance line*
2. $\nexists\, crg \in coreg(m) : l_1 \in crg \wedge l_2 \in crg$
3. *on their instance line,* l_1 *appears* before *(i.e. is drawn closer to the top than)* l_2.

Definition 1.3 (Activated chart) *An* activated chart *is a tuple* $< mode, prech(m), m >$, *where*

- m *is a chart. It is the so-called main chart.*
- $prech(m)$ *is a chart. It denotes the prechart activating* m. *Note that we ignore the particular case of activation messages. We are entitled to do so because they are equivalent to a prechart containing only one event.*
- $mode \in \{nostory, existential, universal\}$.

Note that, by convention, we can use m *to denote the* activated chart $< mode, prech(m), m >$. *However, from the context, it is always clear whether we speak of a main chart or an activated chart.*

Definition 1.4 (LSC specification) *An LSC specification is a set of activated charts* $LS = \{m_1, \ldots, m_n\}$.

1.1.4 Semantics

We give the semantics of our LSC language informally, in terms of *infinite trace languages.*

The semantics for a given chart m is based on the causal order among locations ($<_m$). It is defined as follows:

Definition 1.5 (Causal order ($<_m$)) *We say that* l_1 *and* l_2 *are* directly ordered, *written* $l_1 <_m^d l_2$, *if one of the following conditions holds.*

1. l_2 *is a successor of* l_1 ($l_1 \prec_m l_2$).
2. l_2 *is a successor of* λ *where* λ *and* l_1 *are respectively the send and receive events of the same synchronous message.*
3. l_1 *and* l_2 *are the send and receive events of the same message (synchronous or asynchronous).*

$<_m$ *is the transitive closure of the relation* $<_m^d$.

In order to avoid deadlocks in charts, it is important to ensure that the causal order $<_m$ is acyclic. If it were not, then there would be a location l such that $l <_m l$. This would mean that, before l might be reached, it is necessary that l has already been visited.

Definition 1.6 (Well-formedness) *We say that a chart* m *is* well-formed *if* $<_m$ *is acyclic. We require every chart to be well-formed.*

In a well-formed chart, $<_m$ is a *strict order.*

Definition 1.7 (Cut) *A* cut *through m is a set c of locations, at least one for each instance, such that,*

$$\forall l \in c: \nexists l' \in dom(m) - c : l' <_m l$$
$$\forall \text{ instance } i : \exists l \in c : l \text{ belongs to } i$$

The initial cut *is the cut in which every instance is at its initial location. We write it* $\{0_1, \ldots, 0_n\}$.

c is a final cut *if it enjoys the following property:*

$$\forall l \in c : (\exists l' \in dom(m) - c : l <_m l') \Rightarrow temp(l) \text{ is cold}$$

More intuitively, a cut is a division of the set of locations ($dom(m)$) into two parts: the locations that have already been reached, belonging to the cut c, and the locations that are still to be visited, belonging to $dom(m) - c$.

Example 1.1 (Cut) *In figure 1.2, the locations have been labelled with integers. The cuts of this chart are*

{(Node1,0),(Node2,0),(CS,0)}
{(Node1,0),(Node2,0),(CS,0),(Node2,1)}
{(Node1,0),(Node2,0),(CS,0),(Node2,1),(Node1,1)}
{(Node1,0),(Node2,0),(CS,0),(Node2,1),(Node1,1),(Node1,2)}
{(Node1,0),(Node2,0),(CS,0),(Node2,1),(Node1,1),(Node1,2),(Node2,2)}
{(Node1,0),(Node2,0),(CS,0),(Node2,1),(Node1,1),(Node1,2),(Node2,2),(Node2,3)}
{(Node1,0),(Node2,0),(CS,0),(Node2,1),(Node1,1),(Node1,2),(Node2,2),(Node2,3), (CS,1)}

As a counter-example, note that {(Node1,0),(Node2,0),(CS,0),(Node1,1)} is not a cut, because it is not possible for Node1 to receive a request from Node2 before this message has been sent.

We denote by $cuts(m)$ the set of all the cuts in a chart m.

Now, we will give a dynamic semantics to our language. It will be expressed in terms of *traces* of execution. First of all, we thus need to know how to move from one cut to the next one,

Definition 1.8 (l-successor ($succ_m$)) *We say that a cut c' is a* l-successor *of a cut c in a chart m, written $succ_m(c, l, c')$, if $c \subset c' \ \wedge \ c' - c = \{l\}$.*

Note that LSCs have an *interleaving semantics* because, in one execution step, only one instance is allowed to progress.

We are now able to define a run of an LSC.

Definition 1.9 (Run of a chart m) *A* run *of a chart m is a sequence of cuts $c_0 \cdot c_1 \cdot \ldots \cdot c_k$ satisfying the following requirements:*

- *c_0 is the initial cut of m.*
- $\forall i : 1 \leq i \leq k : \exists\, l \in dom(m) : succ_m(c_{i-1}, l, c_i)$
- *c_k is a final cut*

The trace of a run $r = c_0 \cdot \ldots \cdot c_k$ in m is then defined as

Definition 1.10 (Trace of a run $r = c_0 \cdot \ldots \cdot c_k$) *The* trace *of the run r is the word $e_1 \cdot \ldots \cdot e_k \in Events(m)^*$ where*

$$\forall i : 1 \leq i \leq k : \exists l : \left(\begin{array}{c} succ_m(c_{i-1}, l, c_i) \\ \wedge \\ e_i = ev(l) \end{array} \right)$$

We denote by $Runs(m)$ the set of all the runs in m.

Here comes an important definition, since this concept will be used throughout this whole report,

Definition 1.11 (Trace set of a chart m ($\mathcal{L}_m^{trc}$)) *The* trace set *of a chart m, written $\mathcal{L}_m^{trc} \subseteq Events(m)^*$, is*

$$\mathcal{L}_m^{trc} = \{w \mid \exists r \in Runs(m) s.t.\ w \text{ is the trace of } r\}$$

The next definition finalizes our dynamic semantics of a chart. It is formulated in terms of regular ω-languages over *Events* [2] .

Since LSCs are used to model the behavior of reactive systems, it is natural to use *infinite* sequences of propositions to express their semantics.

As we ultimately want to compare an intra-object specification to an inter-object specification, described by different formalisms, we need to reduce both views of the requirements to a third, common language. We chose to express the semantics of both formalisms in terms of infinite traces. Definition 2.5 uses these ω-languages to define precisely what "a system satisfies its specification" means.

Definition 1.12 (Language of an activated chart m ($\mathcal{L}_m$)) *The language of an activated chart $< mode, prech(m), m >$, written $\mathcal{L}_m \subset Events^\omega$ is*

if $mode = existential$ *then*

$$\mathcal{L}_m = \{\ w_1 \cdot w_2 \cdot \ldots \mid \exists i_1, \ldots, i_n : \left(\begin{array}{l} w_{i_1} \cdot \ldots \cdot w_{i_n} \in \mathcal{L}_{prech(m)}^{trc} \cdot \mathcal{L}_m^{trc} \\ \wedge\ i_1 < i_2 < \ldots < i_n \\ \wedge\ \forall j : i_1 < j < i_n \wedge j \notin \{i_1, \ldots, i_n\} : w_j \notin Events(m) \end{array} \right) \}$$

if $mode = nostory$ *then*

$$\mathcal{L}_m = \{\ w_1 \cdot w_2 \cdot \ldots \mid \nexists i_1, \ldots, i_n : \left(\begin{array}{l} w_{i_1} \cdot \ldots \cdot w_{i_n} \in \mathcal{L}_{prech(m)}^{trc} \cdot \mathcal{L}_m^{trc} \\ \wedge\ i_1 < i_2 < \ldots < i_n \\ \wedge\ \forall j : i_1 < j < i_n \wedge j \notin \{i_1, \ldots, i_n\} : w_j \notin Events(m) \end{array} \right) \}$$

if $mode = universal$ *then*

$$\mathcal{L}_m = \{\ w_1 \cdot w_2 \cdot \ldots \mid \forall i_1, \ldots, i_n : \left(\begin{array}{l} w_{i_1} \cdot \ldots \cdot w_{i_n} \in \mathcal{L}_{prech(m)}^{trc} \\ \wedge\ i_1 < i_2 < \ldots < i_n \\ \wedge\ \forall j : i_1 < j < i_n \wedge j \notin \{i_1, \ldots, i_n\} : w_j \notin Events(m) \end{array} \right) : \exists j_1, \ldots, j_m : \left(\begin{array}{l} w_{j_1} \cdot \ldots \cdot w_{j_m} \in \mathcal{L}_m^{trc} \\ \wedge\ i_n < j_1 < j_2 < \ldots < j_m \\ \wedge\ \forall k : i_n < k < j_m \wedge k \notin \{i_n, \ldots, j_m\} : w_k \notin Events(m) \end{array} \right) \}$$

The first formula states that, for an existential chart, there must be a sequence of events fulfilling the prechart and the main chart in which the restricted events of the chart do not appear.

[2] $\mathcal{L}_m^{trc}$ is a finite language, thus regular. It can be recognized by a finite automaton. This automaton can itself be easily transformed into a *Büchi-automaton* recognizing $\mathcal{L}_m$. The class of languages that can be decided by *Büchi-automata* is called *regular ω-languages*, as stated in [Thomas 90].

The second formula describes the language of a *no-story* chart as the complement of the language of an existential chart. It states that the traces of the chart may never appear in any run.

The third formula states that, for a universal chart, every time the prechart is successfully completed, there must be a sequence of events satisfying the main chart.

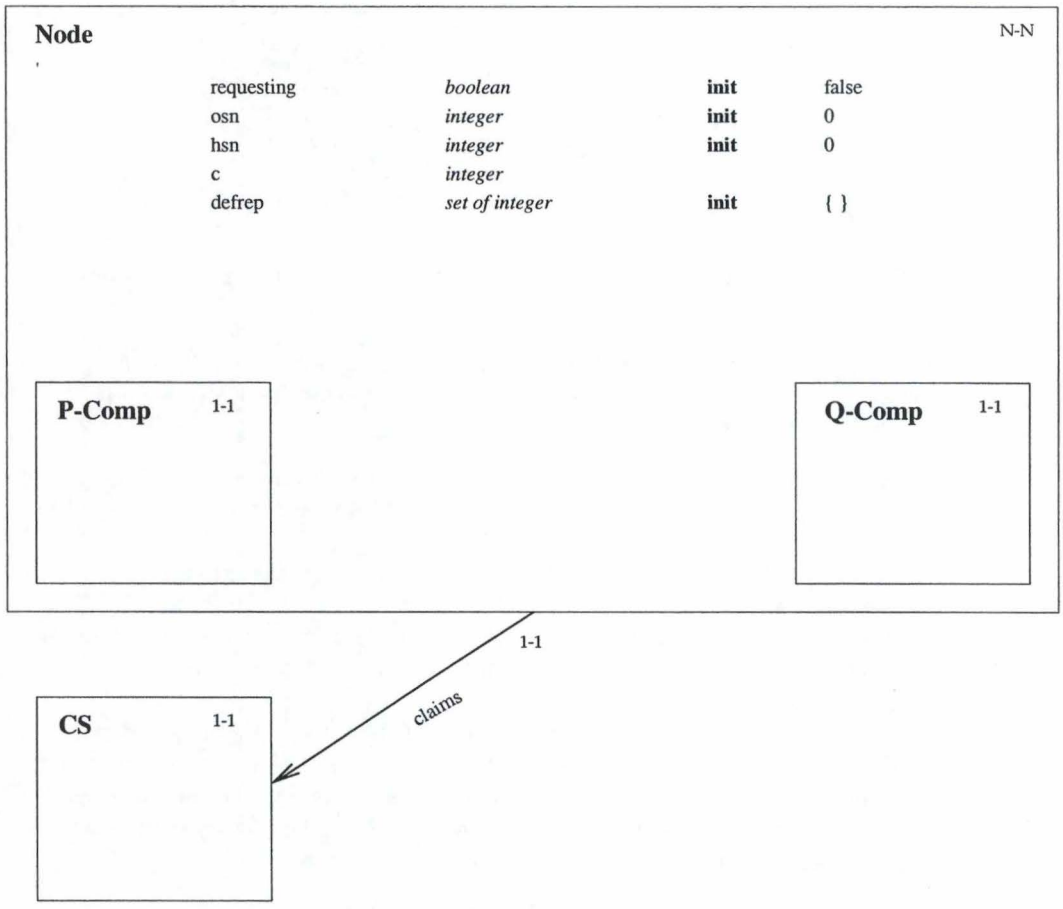


Figure 1.1: Static description of a Ricart-Agrawala system

1.2 An example : Ricart-Agrawala system

Throughout this document, we will use the example of a distributed system to illustrate many concepts. The methods and results presented later have been as much as possible validated on this example.

In this section, we give a description of this system.

It consists of several nodes, competing for a critical section (CS). Figure 1.1 gives a static decomposition of this system into classes, in an Entity-Relationship-like formalism. Here, we respect the assumption of [Harel 00a, p.9], that "the division into objects or components has already been determined".

Each node is a processor, able to run concurrent processes sharing the same memory. We suppose that mechanisms for mutual exclusion are available *within* every node.

There is a logical connection between every node. They communicate by the passing of *asynchronous* messages. The communication is reliable but the order is not guaranteed. This means that if a message is sent, we can only be sure that it will be received after a finite delay, without any alteration to its content. We cannot make any assumption about the delivery order of messages.

We want to achieve *mutual exclusion* on the critical section between all the nodes.

Sender	Recipient	Message
Node	Node	Request(idq,osnq)
Node	Node	Reply
Node	CS	Enter
Node	CS	Exit
env	Node	Ask_CS

Table 1.1: Static specification of messages in Ricart-Agrawala System

Definition 1.13 (Mutual exclusion) *We say that a system achieves* mutual exclusion *on a critical section if it does not allow two distinct nodes to use the critical section at the same time.*

The CS is a passive component, unable to achieve the mutual exclusion by itself. In other words, when a node claims access to the CS, the section may not send back an access denial.

The solution chosen is a distributed solution.*All* the nodes have to agree on the node allowed to enter the CS.

It is *not* a server-oriented solution in which an independent active component would manage the access to the CS and would decide on its own to which node it should grant access .

The messages exchanged between instances of the classes are specified statically on table 1.1 The *request* and *reply* messages are exchanged between nodes to agree on which node is allowed to enter the critical section. The request bears the sender's identifier and a timestamp, in order to prioritize the queries received by a node.

Here are some requirements we have about this system.

Mutual Exclusion The system achieves mutual exclusion. Two nodes can never use the critical section at the same time.

Fairness The system is fair. If a node asks to enter the critical section, it will at last be granted access.

Sensible use of CS When entering the critical section, each node guarantees that it will exit after a finite delay.

Our distributed system will use the well-known *Ricart-Agrawala algorithm* [Sedletsky 00]. The main idea of this algorithm is the following :

> If a node wants to enter the critical section, it must send a *request* to every other node. When it will have received a *reply* from every node, it will be allowed to enter the critical section.

1.2.1 LSCs

In this section, we illustrate the LSC language by showing some examples. They are relevant for a Ricart-Agrawala system instantiated to two nodes and one critical section.

The chart of figure 1.2 means that, every time a node needs to enter the CS, it will follow the protocol prescribed. Moreover, it says that if a node asks to enter the critical section, it will finally be granted access. This corresponds to the *fairness* requirement.

The chart of figure 1.3 corresponds to the *sensible use of CS* requirement. It expresses the constraint that, every time a node enters the critical section, it will eventually exit it.

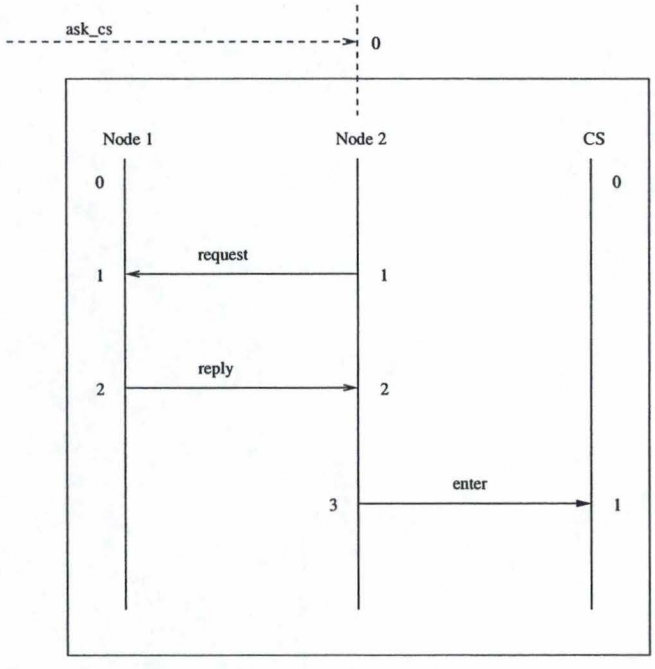


Figure 1.2: No lock-out

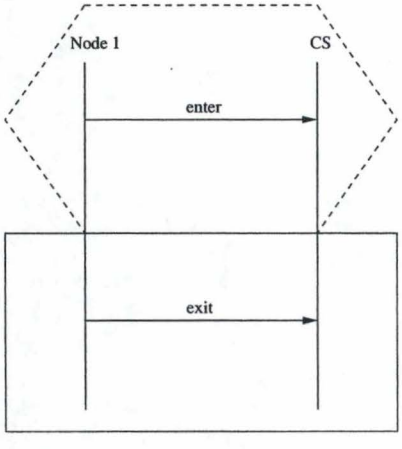


Figure 1.3: No endless loop

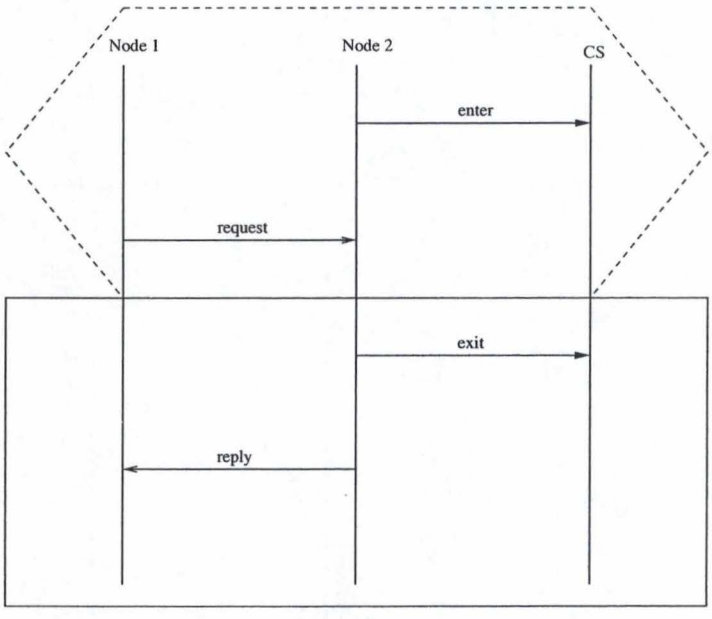


Figure 1.4: Lock the CS

The chart "Lock the CS" (figure 1.4) shows what must happen when a node asks to access the CS while another node is already using it. The node within the critical section must first exit before sending a reply.

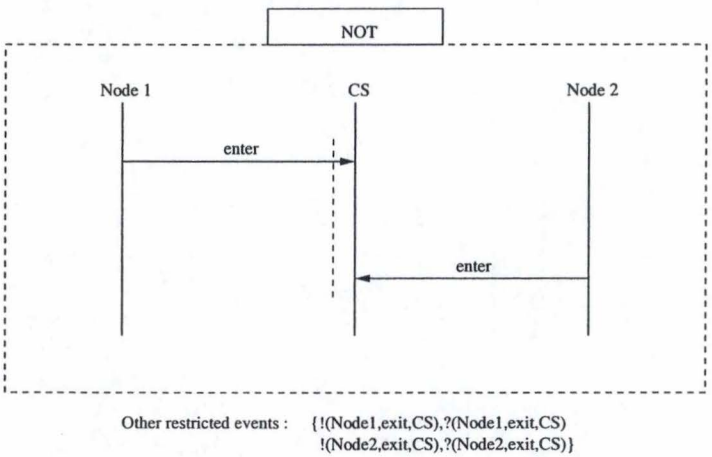


Figure 1.5: Mutual exclusion

The "mutual exclusion" chart (figure 1.5) expresses the *mutual exclusion* requirement. Note that it is a no-story chart, in which some events are restricted, although they are not used in the scenario.

The "crossed requests" charts try to describe how the system should behave when the two nodes send request to each other *at the same time*. Ideally, only one chart might reply.

We wrote "try to describe", because the existential version of this scenario (figure 1.6) does not express correctly that *one* node replies. Indeed, between the second and the third message nothing keeps the second node from entering into the

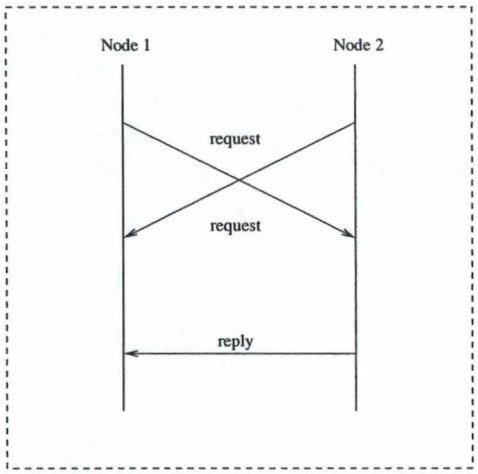
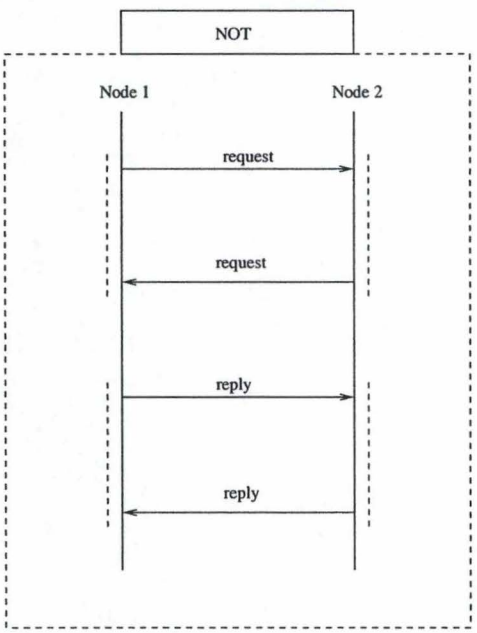


Figure 1.6: Crossed requests (Existential)



Other restricted events : {!(Node1,enter,CS),?(Node1,enter,CS)
!(Node2,enter,CS),?(Node2,enter,CS)}

Figure 1.7: Crossed requests (No-story)

CS, exiting and then replying.

The no-story version of this scenario is much more appropriate (figure 1.7) since it explicitly forbids that the two nodes reply before accessing the critical section.

It is noteworthy that, without the concept of coregions, it would have taken 36 charts to represent the same requirement. Thus, allowing some partiality in the order of events brings a high compactness and significantly improves the readability of the requirements set. Albeit, there is a price to pay for this concision: as explained later, the complexity of the verification process will also be higher.

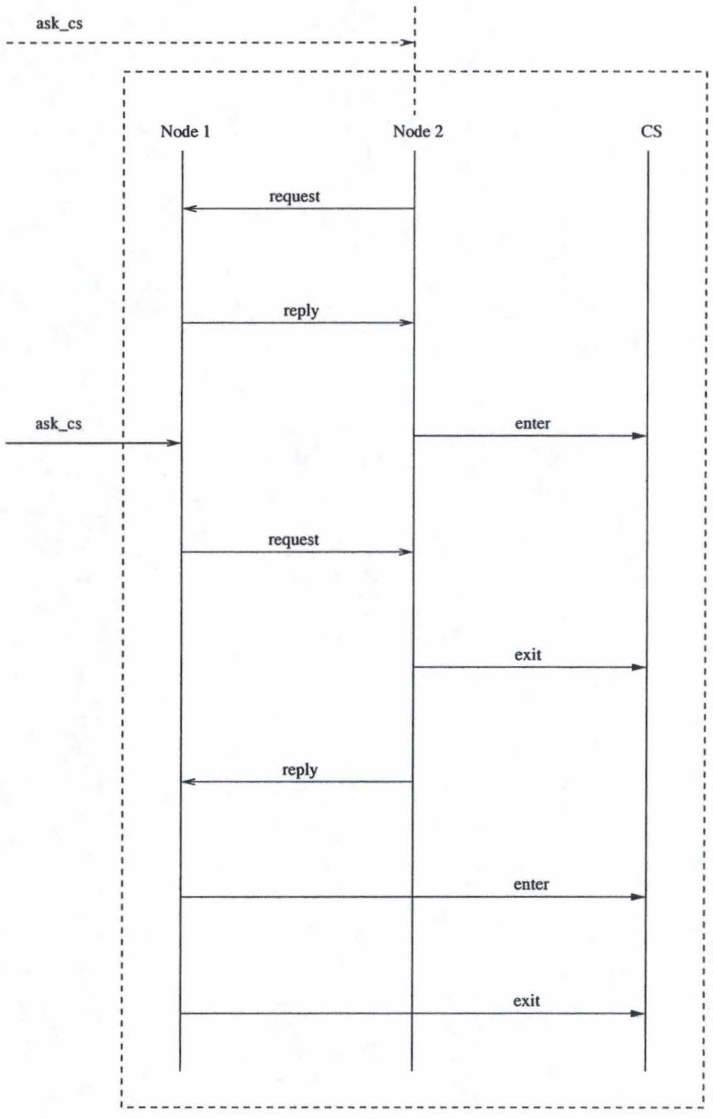


Figure 1.8: Long existential

The scenario of figure 1.8 is just a long existential chart that shows a "complicated" run of the system.

Following the idea proposed in [Harel 98a], the specifiers could think that this existential chart should be promoted into a universal chart, since it seems to exhibit a rather general behavior. So, they might transform it into figure 1.9.

It is not obvious that this leads to an *inconsistency* in the set of scenarios.

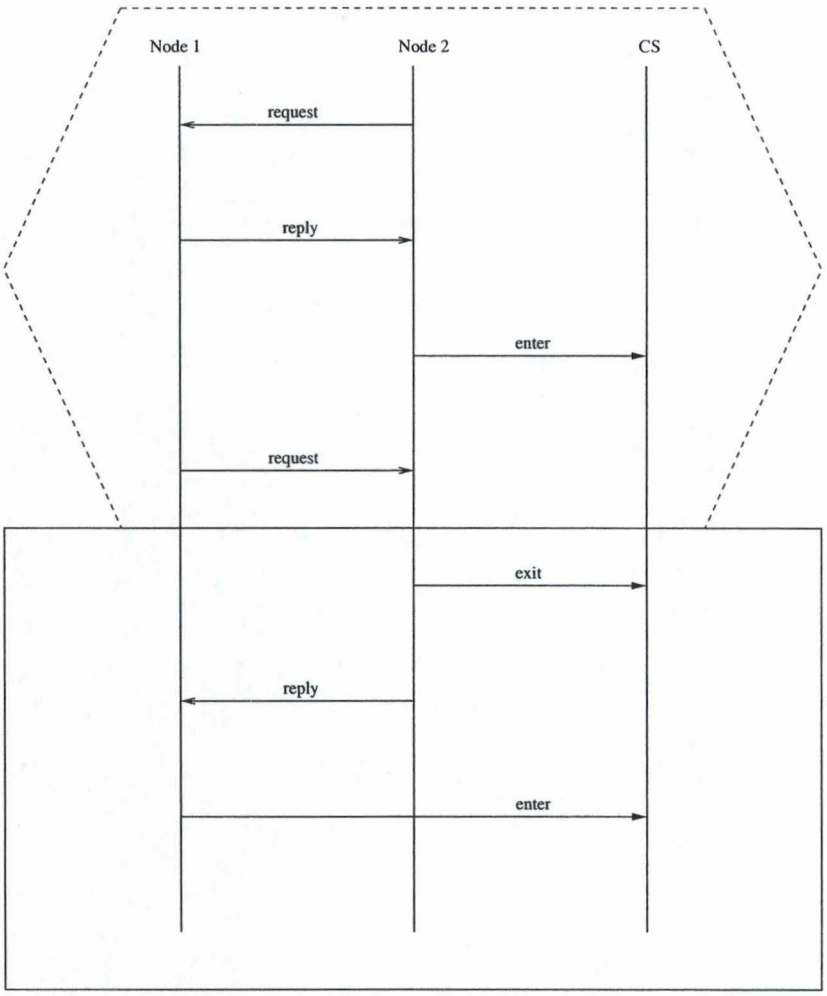


Figure 1.9: Long universal

Indeed, Node2 could request between the moment when it replies to Node1 and the moment when Node1 enters the critical section. This behavior is allowed by the chart 1.4 but is forbidden by the chart 1.9.

This bad promotion or *induction error*[3] illustrates a weak point of the LSC language, namely that the restricted events do not appear clearly in the chart and, consequently, they may lead to inconsistent declarations.

A tool to automatically check the consistency of requirements would be of great help to put up with this problem, see [Harel 00b]. The algorithm 2 presented in this thesis could be a good basis to build such a tool.

An editor providing step-by-step information about restricted events could be helpful, too.

[3]Induce from an example an invalid general rule

Chapter 2

Intra-Object Specification

2.1 Specifying the internal behavior of instances

We have chosen not to impose a particular language to specify the internal behavior of instances.

This hypothesis does not restrict the scope of our work. Actually, it allows it to be even more general than if we had restricted it to a particular language.

Any specification language can use the theory exposed here, provided that a mapping from this language to the Kripke structure formalism can be exhibited. In addition, as requirements expressed as scenarios tell stories about a particular *population* of the system, any specification should provide a means to instantiate it.

It should be noted that the Kripke structure formalism is particularly close to state-based languages for describing internal behavior of objects. Our theory is thus adaptable at a low cost to broadly-used languages, such as statecharts or finite-state machines, for example. These state-based languages are very common in reactive system or hardware control design, see [Gery 96, Harel 99, Selic 94, Patterson 98, Tanenbaum 96]. Anyway, this translation has to be done in a clever way. Otherwise it would result in an exponential blow-up of the number of states, making the verification task practically infeasible, [Chan 01, Chan 99, Staunstrup 00, Derepas 00, Dams 96].

Definition 2.1 (Kripke structure (KS)) *A* Kripke structure *is a tuple* $\mathcal{I} =< S, s_0, T, \pi >$ *where*

- S *is the set of states composing the system.*
- $s_0 \in S$ *is the initial state.*
- $T \subseteq S \times S$ *is the* transition relation *from state to state.*
- $\pi : S \rightarrow 2^{Prop}$ *is an* interpretation function. *Intuitively, it gives, for every states, the propositions that hold in this state.*

Kripke structures are defined in [Dams 96, Kripke 63, Müller-Olm 99, Schmidt 00], and many others. The definitions in all these articles differ on small details, e.g. the interpretation may be considered from states to propositions or from propositions to states (allowing propositions to be *undetermined* in some states) or imposing a set of initial states or not.

Following the same scheme as for the semantics of inter-object specification, we will now define a *run* of a system. This definition will allow us to describe the *language* of the system. Then, the semantics of both formalism will be expressed in the same domain, regular ω-languages over *Events*, and we will be able to compare them, [Manna 95].

Note that we *project* the runs of the KS from the propositions holding in every state onto the propositions in which we are interested, namely the observable events of the system. In other words, we ignore every proposition that is not related with the *communication behavior* of the system.

This abstraction principle can be used in order to improve the verification process by limiting the size of the state space.

Definition 2.2 (Run of a KS) *We say that an infinite sequence of states* $\sigma_0 \cdot \sigma_1 \cdot \sigma_2 \cdot \ldots$ *is a run of a KS* $\mathcal{I} =< S, s_0, T, \pi >$ [1] *if it fulfills the following requirements*

- $\sigma_0 = s_0$

[1] The notation $\mathcal{I}$ stands for $\mathcal{I}$mplementation

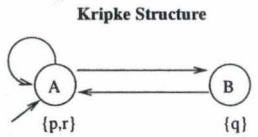


Figure 2.1: A Kripke structure

- $\forall i : \sigma_i \in S$
- $\forall i > 0 : (\sigma_{i-1}, \sigma_i) \in T$

We denote by $Runs(\mathcal{I})$ the set of all possible runs in a KS $\mathcal{I}$.

We require that, in every state, at most one observable event appears. An observable event is a proposition belonging to *Events*. This constraint will be formalized by a temporal logic formula in proposition 3.1 on p.38.

Hypothesis 2.1 (Mutually exclusive events in a KS $\mathcal{I}$) *We say that a KS $\mathcal{I}$ ensures that observable events are mutually exclusive if, in every run $r = \sigma_0 \cdot \sigma_1 \ldots \in Runs(\mathcal{I})$,*

$$\forall\, i \;:\; |\pi(\sigma_i) \cap Events| \leq 1$$

This hypothesis requires that, in every step, at most one observable event appears. This is needed in order to be able to deduce a total ordering of observable events from a sequence of states visited by a run. If we allowed several events to occur in the same state, we would have some trouble comparing this run with the run of a chart (definition 1.9), in which we assumed that events appeared in distinct moments.

Definition 2.3 (Trace of a run) *If $r = \sigma_0 \cdot \sigma_1 \ldots$ is a run in a KS $\mathcal{I}$, then, its* trace, *written $trace(r) \subseteq Events^\omega$, is defined as*

$$f(\sigma_0) f(\sigma_1) \ldots$$

where

$$f(\sigma) = \begin{array}{llll} if & \pi(\sigma) \cap Events = \{e\} & then & e \\ else & \pi(\sigma) \cap Events = \emptyset & & \bot \end{array}$$

We only keep observable events. When the run goes through a state in which no observable event appears, the trace reports the occurrence of $\bot$ (false).

Definition 2.4 (Language of a KS) *The language of a given KS $\mathcal{I}$, written $\mathcal{L}_\mathcal{I} \subseteq Events^\omega$ is defined as follows*

$$\mathcal{L}_\mathcal{I} = \{w \mid \exists r \in Runs(\mathcal{I}) \;:\; w = trace(r)\}$$

Example 2.1 (Kripke structure) *Figure 2.1 presents a graphical representation of the Kripke structure*

$$< \{A, B\}, A, \{(A, A), (A, B), (B, A)\}, \{A \mapsto \{p, r\}, B \mapsto \{q\}\} >$$

Its language is $(pr[q] \mid rp[q])^\omega$.

2.2 Satisfaction

We are now able to define the central concept of our work, the *satisfaction* of a set of scenarios by a system.

Definition 2.5 (Satisfaction) *We say that a system specified by a Kripke Structure $\mathcal{I}$* **satisfies** *an LSC specification LS if* $\forall < mode(m), prech(m), m > \in LS$,

$$\begin{array}{ll} \text{if } mode(m) = universal, & \mathcal{L}_{\mathcal{I}} \subseteq \mathcal{L}_m \\ \text{if } mode(m) = existential, & \mathcal{L}_{\mathcal{I}} \cap \mathcal{L}_m \neq \emptyset \\ \text{if } mode(m) = nostory, & \mathcal{L}_{\mathcal{I}} \subseteq \mathcal{L}_m \end{array}$$

As already emphasized, the definition for a no-story is nothing but the negation of the definition for an existential chart.

2.3 An example of intra-object specification

Below, we propose an intra-object specification in a language similar to the *Extended Finite State Machine* language for our example distributed system [Patterson 98].

We did not specify it using the KS formalism since it would have yielded a mostly unreadable and lengthy specification.

Instances are identified by integers. $C[i]$ denotes the i-th instance of class C. $m(p) \rightarrow C[i]$ means that the current instance sends a message m with actual argument p to the i-th instance of class C. *self* is a variable containing the identifier of the current instance.

For the sake of brevity, we do not describe the syntax and semantics of this language here but rely instead on the intuition of the reader.

In appendix B to this document, we give the SMV code for a Ricart-Agrawala system instantiated to two nodes. As it is stated in [McMillan 00], "the primary purpose of the SMV input language is to describe Kripke structures". This example shows that our specification can be translated into a Kripke structure. This implementation served as a testbed for the methods and results presented in this report. These experimental results are summarized in chapter 6.

2.3.1 Ricart-Agrawala system

A Ricart-Agrawala system is composed of several nodes, concurrently accessing a critical section. We want to achieve mutual exclusion on this critical section.

The nodes communicate by the passing of two asynchronous messages: *reply* and *request*, in order to agree on the node that will be allowed to enter the critical section. Request messages are labelled with the sender's identifier and a timestamp. These are used to determine, upon the reception of a message, the priority of the query.

The Node class

Every node has three components: a main component, that reacts properly to user's stimuli to access the critical section, a p-component, which senses the communication channel and counts the received replies, and a q-component, which senses the communication channel and replies to other nodes' queries.

As described in figure 1.1, a node has several variables:

- *requesting*, a flag to determine if the node has begun requesting for the critical section, but has not yet been granted access;

- *osn*, the own sequence number of the node, which contains the timestamp that labelled the last request sent by the node;
- *hsn*, the highest sequence number received so far by the current node;
- *c*, a counter for the number of replies still to be received;
- *defrep*, the deferred replies, i.e. replies that are to be sent as soon as possible by the current node.

The behavior of a node is described by the EFSM of figure 2.2

Upon user's request, i.e. the arrival of *ask_cs*, the node becomes requesting. It follows a first transition to enter state $N2$. This initializes the *requesting* flag to true, *c*, the counter of replies to be received, to the number of nodes in the system, minus one (the current node), and gives a fresh timestamp to its request, by incrementing its highest sequence number.

Then, it sends a request labelled by its identifier (*self*) and the chosen timestamp, to every other node in the system and waits in state $N3$.

When the node has received a reply from all the nodes in the system, it knows that entering the critical section is safe. Hence, when $c = 0$, it may enter the critical section, and follows the transition leading to state $N4$.

The node eventually exits the critical section. Since it is no more requesting, it turns the *requesting* flag to false and moves to state $N5$.

Because the node might have received several requests while it was using the critical section, it now needs to send these delayed answers, which leads the node to state $N6$ and, then, to the idle state $N1$.

The P-Component class

This component is just a module reacting to the reception of a reply from other nodes. When a reply is received, it decrements the *c* counter.

This object behaves as prescribed in figure 2.3

The Q-Component class

The *q*-component decides, upon the reception of a request, if the node should reply immediately or delay its answer, see figure 2.4.

Clearly, the answer depends on the requesting state of the node. If the node is not interested in the critical section, i.e. its *requesting* flag is set to false, it will immediately reply to every request it receives. However, the component should still check that *hsn* is always the highest sequence number of every received request. Thus, if it receives a request labelled with a more recent timestamp, it modifies *hsn*. This behavior is that of the transitions emanating from state $Q3$.

If the node is requesting, its answer depends on the age of the request. If the node receives a request that is older that its own request, it considers that this query has a higher priority and thus, replies. If the received query is younger than its own, it stores the sender's identifier in its deferred replies list.

When the node receives a request that is exactly as old as its own request, in other words, that bear the same timestamp *osnq* as *osn*, the identifier is used to decide which node will have the highest priority. As a convention, a node with a smaller identifier will be granted access, when such a conflict arises.

The Critical Section

In our intra-object specification, we chose to explicitly model the critical section as a passive object, receiving two kinds of messages: *enter* and *exit*.

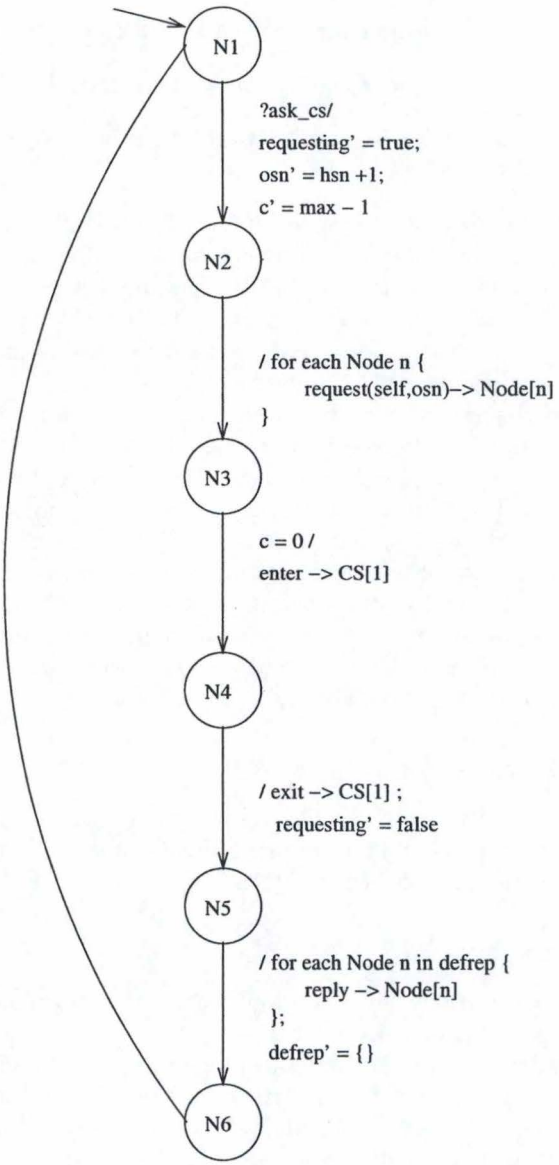


Figure 2.2: EFSM for the *Node* class

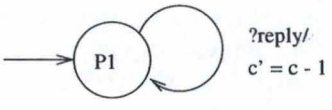


Figure 2.3: EFSM for the *P-Comp* class

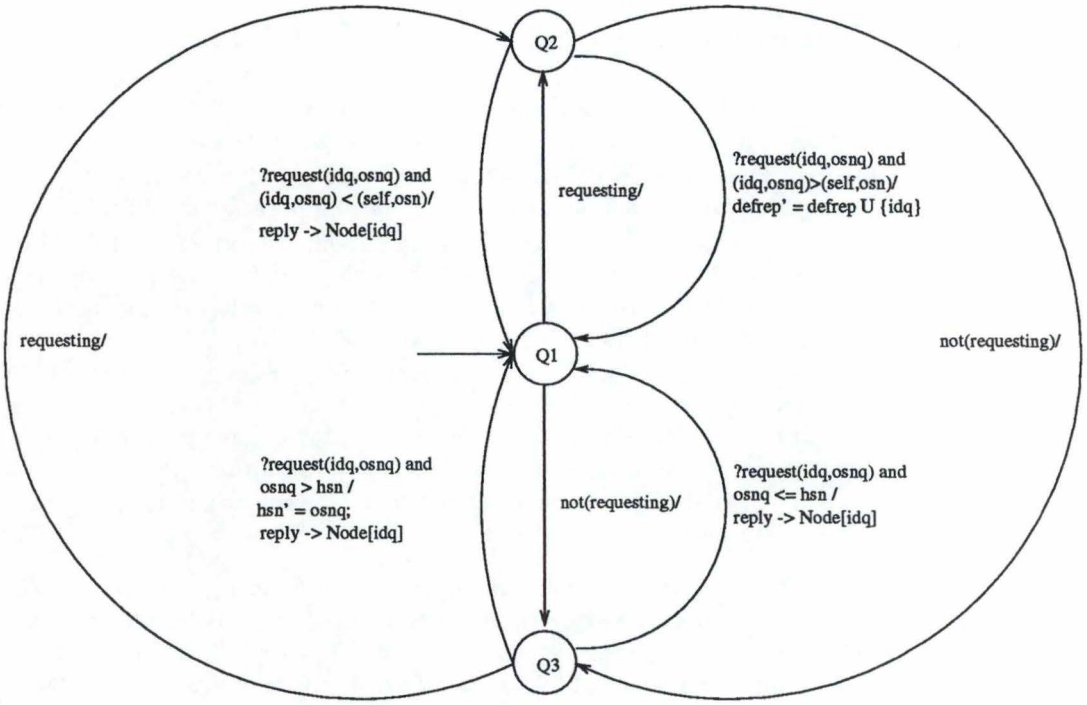


Figure 2.4: EFSM for the *Q-Comp* class

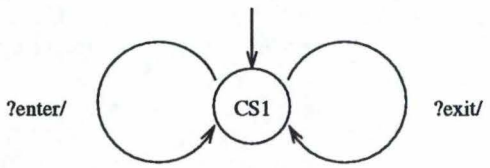


Figure 2.5: EFSM for the *CS* class

It is noteworthy that this object, as it is model in figure 2.5, respects our primary assumption of page 16 that it has no intelligence and is thus unable to decide on its own which node should be allowed to enter.

2.4 Temporal logic

For an in-depth survey of modal logics, refer to [Emerson 90]. The introductory part of this section has been largely inspired from this excellent article.

[Manna 95, Schmidt 00, Müller-Olm 99] also provide good introductions to other kinds of temporal logics (modal mu-calculus, linear temporal logic and CTL).

In classical logic, formulae are used to reason about the present world. Modal logics were developed to allow expression of possibility. For example, an assertion P may be false in the present world but the assertion *possibly* P may be true if there exists an alternate world fulfilling the property P.

Temporal Logic is a particular class of modal logics. It is used to reason about how the truth value of an assertion varies with time. Typical temporal operators include *sometime* P which is true now if there is a future moment in which P will be true and *always* P which is true now if P is true in all future moments.

The logic we use in this report is *propositional*, *branching-time*, *point-based*, *discrete* and *future-tense*.

In *propositional* TL, the non-temporal portion of the logic is just classical propositional logic, i.e. possibly negated disjunctions and conjunctions of atomic propositions. Its counterpart is *higher-order* TL, in which the propositions of propositional TL are refined into expressions built up from variables, functions, predicates and quantifiers.

When defining a system of temporal logic, there are several possible views regarding the underlying nature of time. One is that the course of time is *linear*. At each moment, there is only one future moment. The opposite one is that the course of time is *branching* : at every moment, time may split into different courses representing alternate futures.

A point-based logic is a logic in which the assertions are evaluated in *points* in time. Another view would be having temporal operators that are evaluated over *intervals* of time.

We consider the time as *discrete*. It means that the different moments may be mapped onto the set of nonnegative integers. *Continous* logic is another approach to time. In this case, the moments are mapped onto the real numbers.

We are only interested in *future* properties. We will consider the present as the moment in which the system is started. In the future, we want the system to exhibit the behavior prescribed in the scenarios. We are not interested in the history of the system, i.e. properties about the moments preceding the present, which are the kind of problems *past operators* deal with.

2.4.1 CTL* (Computational Tree Logic)

Tree-like structures

In branching-time logics, the underlying structure of time is assumed to have a branching tree-like structure where each moment may have many successor moments.

Kripke structures specify a non-deterministic behavior. The possible runs of a KS can thus be seen as a tree-like execution. In each state, the system may *choose* the next one among several states. So, there are many *future moments* for a given present.

This is represented in figure 2.6.

Figure 2.6: A Kripke structure induces a tree-like execution.

Syntax

$$
\begin{array}{lll}
\text{state formula} & ::= & p \in Prop \\
& | & < \text{state formula} > \ \wedge \ < \text{state formula} > \\
& | & \neg \ < \text{state formula} > \\
& | & E \ < \text{path formula} > \\
& | & A \ < \text{path formula} > \\
 & & \\
\text{path formula} & ::= & < \text{state formula} > \\
& | & < \text{path formula} > \ \vee \ < \text{path formula} > \\
& | & X \ < \text{path formula} > \\
& | & < \text{path formula} > \ U \ < \text{path formula} > \\
& | & G \ < \text{path formula} > \\
& | & F \ < \text{path formula} >
\end{array}
$$

The set of *path formulae* forms the language CTL*.

Semantics

A *state formula* describes a property about a moment (or state) of the execution. This can be a simple assertion, i.e. a formula in propositional logic. Since, from one state, there are many possible executions, two *path quantifiers* are provided. $E\ p$ means that, from this state, *at least one* execution must exhibit the property p, while with $A\ p$, we require that *every* execution emanating from this state fulfills p.

A *path formula* states a property about one execution, i.e. one sequence of states, of the system. The operators X and U stand for *Next* and *Until*. $X\ p$ means that in the next state of the path considered, the property p holds. $q\ U\ p$ is read "q Until p" and means that, on the path considered, there will eventually be a state s satisfying p and every state preceding s must satisfy q.

G and F are read *Always* and *Finally*. They mean that, on a path, a property must be true in every (resp. at least one) state.

These two operators can be expressed in terms of U, knowing that

$$
\begin{array}{lcl}
G\ p & \iff & \neg F \neg p \\
F\ p & \iff & true\ U\ p
\end{array}
$$

Definition 2.6 (Satisfaction) *We give the interpretation of these formulae with respect to a KS* $\mathcal{I} =< S, s_0, T, \pi >$. *Let* $s_0 s_1 \ldots \in Runs(\mathcal{I})$. *We use three meta-variables, p, an atomic proposition,* σ, *a state formula and* ϕ, *a path formula.*

We write $\mathcal{I}, s \models \sigma$ *to say that the state s satisfies the* state formula σ *in the system* $\mathcal{I}$ *and* $\mathcal{I}, (s_0 s_1 \ldots) \models \phi$ *to say that the run* $(s_0 s_1 \ldots)$ *satisfies the* path formula ϕ *in the system* $\mathcal{I}$.

$$
\begin{array}{lcl}
\mathcal{I}, s \models p & \Longleftrightarrow & p \in \pi(s) \\
\mathcal{I}, s \models \neg\sigma & \Longleftrightarrow & \mathcal{I}, s \not\models \sigma \\
\mathcal{I}, s \models \sigma_1 \wedge \sigma_2 & \Longleftrightarrow & \mathcal{I}, s \models \sigma_1 \textbf{ and } \mathcal{I}, s \models \sigma_2 \\
\mathcal{I}, s \models A\,\phi & \Longleftrightarrow & \forall r \in Runs(< S, T, s, \pi >) : \mathcal{I}, r \models \phi \\
\mathcal{I}, s \models E\,\phi & \Longleftrightarrow & \exists r \in Runs(< S, T, s, \pi >) : \mathcal{I}, r \models \phi \\
 & & \\
\mathcal{I}, (s_0 s_1 \ldots) \models \sigma & \Longleftrightarrow & \mathcal{I}, s_0 \models \sigma \\
\mathcal{I}, (s_0 s_1 \ldots) \models \phi_1\ U\ \phi_2 & \Longleftrightarrow & \exists i \geq 0 : \mathcal{I}, (s_i s_{i+1} \ldots) \models \phi_2 \\
 & & \textbf{and } \forall j : 0 \leq j < i : \mathcal{I}, (s_j s_{j+1} \ldots) \models \phi_1 \\
\mathcal{I}, (s_0 s_1 \ldots) \models F\ \phi & \Longleftrightarrow & \exists i \geq 0 : \mathcal{I}, (s_i s_{i+1} \ldots) \models \phi \\
\mathcal{I}, (s_0 s_1 \ldots) \models G\ \phi & \Longleftrightarrow & \forall i \geq 0 : \mathcal{I}, (s_i s_{i+1} \ldots) \models \phi \\
\mathcal{I}, (s_0 s_1 \ldots) \models X\ \phi & \Longleftrightarrow & \mathcal{I}, (s_1 s_2 \ldots) \models \phi
\end{array}
$$

Note that, in the definition of E and A, the Kripke Structure used in the right-hand expression has changed, since its initial state is now s, instead of s_0.

2.4.2 Models

Definition 2.7 (Model) *We say that a KS* $\mathcal{I}$ *is a* model *of a CTL* state formula* σ, *and we write it* $\mathcal{I} \models \sigma$, *if for every run* $r \in Runs(\mathcal{I})$

$$\mathcal{I}, r \models \sigma$$

In the remainder of this document, when $\mathcal{I}$ is understood from the context, we will simply write $r \models p$ instead of $\mathcal{I}, r \models p$.

Definition 2.8 (Local model-checking) *The* local model-checking problem *is defined as follows :*

> *Given a KS* $\mathcal{I}$ *and a formula* φ, *determine if* $\mathcal{I}$ *is a model of* φ, *i.e.* $\mathcal{I} \models \varphi$, *[Dams 96, Müller-Olm 99].*

This problem is qualified as *local*, because the initial states are fixed. The problem is said to be *global* if the runs on which we try to check φ may start from *any* state in $\mathcal{I}$.

We have now introduced all the concepts needed in order to justify our choice of using model-checking to verify the satisfaction of a set of scenarios by a system.

On the one hand, model-checkers provide a *push-button* approach to prove that a system satisfies a desired property. This full automation together with the fact that this field is quickly evolving, yielding increasingly *efficient* tools, form their attractiveness.

On the other hand, the property that *LSCs can be embedded into the CTL* language* has been exhibited in [Harel 00b, Kugler 01] where general formulae equivalent with charts have been presented. However, these formulae only applied to a subset of LSCs more restrictive than ours. In this report, we will thus need to extend them to our language. So, to verify that a system $\mathcal{I}$ satisfies a chart m, we can translate m into a CTL* formula φ_m and check if $\mathcal{I} \models \varphi_m$.

Moreover, our choice to use *Kripke structures* as a specification language was not innocent. As we already emphasized, it belongs to the class of *transition systems*, and broadly-used state-based specification languages translate easily into this formalism. For instance, Statecharts [Gery 96, Harel 98b] are used in a lot of specification methods, among which UML, where they are called *StateDiagrams* [Jacobson 99]. State machines are also used in the ROOM method to specify objects behavior, see [Selic 94], or hardware control, see [Patterson 98]. Telecommunication protocols are often partially specified using this kind of formalism [Tanenbaum 96]. It is also a unifying framework for model-checking. Consequently, a specification expressed in KS (or any equivalent form) can be used almost without any modification as input to model-checkers.

[Chan 01, Chan 99] describe techniques to optimize the model-checking of properties on Statecharts specifications. [Alur 98] also presents means to cope with model-checking problem on high level description languages.

Hence, it would be sufficient to find ways to *automatically* translate charts into CTL* formulae to allow a full automation of the verification.

In the following chapter, we will present an algorithm to perform this automatic translation. Knowing that the model-checking problem is a complex problem requiring much time and space resources, we will then present some means to optimize the verification

Chapter 3

Automating the Translation of LSCs into CTL*

3.1 Structure of the algorithm

We propose a structure for the automatic translation, decomposing it into several subproblems. Figure 5.1, on page 85 presents an LSC summarizing graphically how these different subproblems interact.

3.1.1 Build the causal order

Build the causal order $<_m$ among all locations of m. We already defined this partial order, in definition 1.5.

3.1.2 Build $A_{cuts(m)}$ for a given chart m

Build an automaton $A_{cuts(m)}$, such that $\mathcal{L}(A_{cuts(m)}) = \mathcal{L}_m^{trc}$. The following characterization of this automaton is inspired from [Harel 00b].

Definition 3.1 ($A_{cuts(m)}$) *$A_{cuts(m)}$ is an automaton $< A, S, s_0, \rho, F >$. where,*

- *A is the alphabet of the automaton. Here, $A = Events(m)$, the set of events restricted in* m.
- *S is the set of states of the automaton. Here, $S = cuts(m)$, the set of cuts in* m.
- *s_0 is the initial state. Here, $s_0 = \{0_1, \ldots, 0_n\}$, the initial cut (every instance is at its initial location).*
- *$\rho \subseteq (S \times A \times S)$ is the transition relation where*

$$\rho(c, e, c') \iff succ_m(c, loc, c') \wedge e \text{ is the event associated with } loc$$

- *F is the set of final states. Here, $F = \{c \mid c \text{ is a final cut in } m\}$. This notion has been introduced in definition 1.7.*

By its definition, one can see that this automaton is non-deterministic. By the fact that its language is finite, it is also acyclic and finite. We abbreviate *acyclic non-deterministic finite automata* by *ANFA*.

We state two properties of these automata, making it possible to represent them as directed graphs.

Property 3.1 (Retrievability of transition labels) *Knowing that $(c, e, c') \in \rho$, we have, by the definition of ρ, that e is the event associated with the location loc, where $succ_m(c, loc, c')$. Referring to the definition 1.8 (l-successor), we get $\{loc\} = c' - c$.*

Theorem 3.1 (Uniqueness of transitions) *The following comes as a corollary of property 3.1*

$$\forall c, c' \in S : \nexists e, e' \in Events(m) : (e \neq e') \wedge ((c, e, c') \in \rho) \wedge ((c, e', c') \in \rho)$$

Proof 3.1 (Theorem 3.1) *Indeed, if we suppose that there are two transitions labelled by distinct events e and e' between the cuts c and c', we know that both e and e' are the events associated with loc, where $\{loc\} = c' - c$ (property 3.1). Immediately, we deduce that $e = e'$. Hence, we reach a contradiction with our primary hypothesis and we have to conclude that there is at most one transition between c and c'.*

□

Note that the reverse

$$(c, e, c') \in \rho \wedge (c, e, c'') \in \rho \Rightarrow c' = c''$$

is not necessarily true. As a counter-example, consider the case of two identical messages sent in the same coregion.

These properties allow us to use a simpler representation for $A_{cuts(m)}$, when needed. Instead of using an ANFA, we can use a directed graph, (V, E).

$V = cuts(m)$.

$E \subseteq V \times V$, where $(c, c') \in E \iff \exists loc \in dom(m) : succ_m(c, loc, c')$.

Note that it is not necessary to retain s_0 and F, since, by the characterization of these elements, we can decide which states in V are initial and final. The initial state is the initial cut and a final cut corresponds to an accepting state.

Using this graph-oriented representation, the following theorem is easily shown.

Theorem 3.2 *Let $c, c' \in cuts(m)$ be two distinct cuts. Then if $\exists c'' \in cuts(m)$ such that $(c, e, c'') \in \rho$ and $(c', e', c'') \in \rho$, with $e \neq e'$, we have that*

$$\begin{array}{c} (\exists\ l\ :\ (l \in c') \wedge (e = ev(l))) \\ \wedge \\ (\exists\ l'\ :\ (l' \in c) \wedge (e' = ev(l'))) \end{array}$$

Proof 3.2 (Theorem 3.2) *We use the retrievability property of labels. If we suppose that both (c, e, c'') and $(c', e', c'') \in \rho$, then, there are l, l' such that $c'' - c = \{l\}$ and $c'' - c' = \{l'\}$, where e, e' are the events associated with l, l' respectively. Obviously, since $c'' = c \cup \{l'\} = c' \cup \{l\}$, it comes that $l \in c'$ and $l' \in c$.*

□

This theorem will be used later on, when we will apply a semantical transformation to LSCs.

3.1.3 Assemble the main chart and its prechart

Build $A_{append(m)}$ such that $\mathcal{L}(A_{append(m)}) = \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m}$.

3.1.4 Generate the formula from the trace set

First of all, we note that trace sets as they have been defined above are necessarily finite. Intuitively, this is deduced from the fact that the locations are finite. Because the cuts are contained in $2^{dom(m)}$, there is a finite number of cuts. The transitions between the cuts define the words recognized by the chart and these transitions cannot induce cycles, since the chart is well-formed. This conclusion comes directly when considering propostion 3.1.

Secondly, we define a binary relation on $Events(m)$, to express the constraint that the semantics of LSCs imposes that *observable events are mutually exclusive*, see hypothesis 2.1.

Definition 3.2 (Mutually exclusive events in a system $\mathcal{I}$) *We say that two events e_1, e_2 are* mutually exclusive in a system $\mathcal{I}$, *written $e_1 \nmid e_2$, iff*

$$\mathcal{I} \models AG(\neg(e_1 \wedge e_2))$$

In the following sections, we will make the assumption that **the relation $\nmid$ is *total***, i.e.

Hypothesis 3.1 (Mutual exclusion between events)

$$\forall \mathcal{I} \; : \; \forall \, e_1, e_2 \; \in \; Events(m) \; : \; e_1 \nmid e_2$$

It means that, in any system considered, in any state of execution, at most one event can appear.

Definition 3.3 (ϕ_w) *Let $\phi_w \in CTL^*$, with $w \in Events(m)^*$ be defined as*

$$\phi_{e_1 \cdot e_2 \cdots e_q} \; = \; (e_1 \wedge X(N \, U \, (e_2 \wedge X(N \, U \, (e_3 \wedge X(\ldots X(N \, U \, e_{n-1} \wedge X(N \, U \, e_q) \ldots)$$

$$\textit{where } N \; = \bigwedge_{e \, \in \, Events(m)} \neg e$$

Given the chart m as input, we want the algorithm to produce φ_m, the CTL* formula expressing the scenario described by m. [Kugler 01, Harel 00b]

Definition 3.4 (CTL* formula for a universal chart m)

$$\varphi_m \; = \; AG(\bigvee_{w \, \in \, \mathcal{L}^{trc}_{prech(m)}} \phi_w \quad \rightarrow \quad \bigvee_{w \, \in \, \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m}} \phi_w)$$

Definition 3.5 (CTL* formula for an existential chart m)

$$\varphi_m \; = \; EF(\bigvee_{w \, \in \, \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m}} \phi_w)$$

These formulae are built taking as a parameter the *trace set* of the chart. In the universal case, for example, we will have one disjunct for every possible sequence of events. In this stage of the algorithm, we have already dealt with the issue of asynchronism of messages or coregions. This problem has been tackled at the very first stage, in which we built the causal order $<_m$ among all locations of m.

The consequence of introducing the concepts of coregions or asynchrony is that they *release* the constraint on the ordering of events, as the axioms (definition 1.5) defining this order clearly show. So, they induce possible permutations between events and hence, alternative sequences of events. In summary, these two notions *add* words to the trace set of the chart.

3.2 Development of the algorithm

3.2.1 Creation of $<_m$

Using the axioms defining $<_m$ and the information of the abstract syntax, this step is straightforward. However, an efficient representation of the order would have a very positive impact on the performance of the later steps of the algorithm, in which it will be heavily used.

It is interesting to check the well-formedness (def. 1.6) of the chart at this level. To do so, we propose the fix-point algorithm below, which builds, for a given partial

order $<$, the maximum set of elements such that the transitive closure of $<$, which we write $<^*$, is acyclic. More formally, for a binary relation $<$ on S, this algorithm outputs the set $Acyclic = \{e \in S | \neg(e <^* e)\}$ and $\forall e \in S - Acyclic : e <^* e$.

Consequently, to check that $<_m$ is acyclic, we use algorithm 1 on $<_m^d$ and we test if $dom(m) - Acyclic = \emptyset$.

Algorithm 1 Is $<^*$ acyclic ?

Input : a set of element E and a partial order on this set $<$.
Output : *Acyclic*, the maximum subset of E such that

$$\forall e \in Acyclic : \neg(e <^* e)$$

{Init}
$Acyclic' \leftarrow \emptyset$

{Iter}
repeat
  $Acyclic \leftarrow Acyclic'$
  $Acyclic' \leftarrow Acyclic \cup \{e | \{e' | e' < e\} \subseteq Acyclic\}$
until $Acyclic' = Acyclic$

3.2.2 Creation of $A_{cuts(m)}$

We will construct the automaton iteratively, from the initial cut to all the final cuts, basing this construction on the *causal order*, $<_m$.

This algorithm is quite similar to the idea briefly exposed in [Alur 99], although it had been developed independently.

The solution to this subproblem is indeed the core of our algorithm for automatic translation. With some post-processing of the output, it could be used for the *synthesis* of specifications from LSCs, see [Harel 00b].

We use a new function, $events : 2^{dom(m)} \rightarrow 2^{Events(m)}$, mapping a set of locations to the set of events relative to them.

Algorithm's data structure

The algorithm will use the following main variables.

1. $\mathbb{Q} \subseteq cuts(m)$ – The states still to be developed.
2. $\mathbb{S} \subseteq cuts(m)$ – The states of $A_{cuts(m)}$ already found.
3. $\mathbb{F} \subseteq \mathbb{S}$ – The final states in $\mathbb{S}$.
4. $\rho \subseteq (\mathbb{S} \times Events(m) \times \mathbb{S})$ – The transition relation on $\mathbb{S}$.

And work variables,

5. c – The element of $\mathbb{Q}$ chosen to be developed.
6. $L_\perp$ – The set of *smallest* events (according to $<_m$) in $dom(m) - c$.
7. c' – The successor of c (every instance is at its previous location, excepted one that moved forward by one location).

This algorithm requires m to be a well-formed chart.

The **postcondition** of this algorithm is that the automaton $< Events(m), \mathbb{S}, \rho, \{0_1, \ldots, 0_n\}, \mathbb{F} >$ recognizes the words belonging to $\mathcal{L}_m^{trc}$ and only those.

Invariant

Intuitively, the invariant (*Inv*), can be formulated as follows.

"A part of $\mathrm{A}_{cuts(m)}$ has already been built, we call it A_S. For every word in the trace set of m, A_S is able to recognize either the whole word or a prefix of it, assuming pending states are final. If it is only able to recognize a prefix, it leads to a state about which enough information is kept in $\mathbb{Q}$ to also recognize the suffix. The partial automaton does not recognize any other word than those in the trace set of m, or prefixes of them."

This proposition is formally formulated in figure 3.1.

$$\text{Let} \quad A_S = < Events(m), \mathbb{S}, (0, \ldots, 0), \rho, \mathbb{Q} \cup \mathbb{F} > .$$

$$\forall\, w \,:\, w \in \mathcal{L}_m^{trc}$$

$$\Longleftrightarrow$$

(Inv_1) :

$$\left(\begin{array}{l}
\exists\, \mathbb{S}', \rho', \mathbb{F}', A_{\mathbb{Q}} \text{ such that} \\
\mathbb{S}' = \mathbb{Q} \cup (cuts(m) - \mathbb{S}) \\
\wedge \\
\rho' = (c, e, c') \Longleftrightarrow \left(\begin{array}{l} succ_m(c, loc, c') \wedge \\ e = ev(loc) \end{array}\right) \\
\wedge \\
\mathbb{F}' = \{c \mid c \text{ is a final location in } m \wedge c \in \mathbb{S}'\} \\
\wedge \\
\mathrm{A}_Q = < Events(m), \mathbb{S}', \mathbb{Q}, \rho', \mathbb{F}' > \\
\wedge \\
\quad w \text{ can be decomposed into } w_1 \cdot w_2 \text{ such that} \\
\left[\begin{array}{c} w_1 \in \mathcal{L}(\mathrm{A}_S) \text{ and leads to a final state } c \in \mathbb{Q} \\ \Downarrow \\ \left(w_2 \text{ is recognized by } \mathrm{A}_Q \text{ from the initial state } c \wedge w_2 \in events(dom(m) - c)^*\right) \end{array}\right]
\end{array}\right)$$

$$\vee$$

(Inv_2) :

$$\left(w \in \mathcal{L}(\mathrm{A}_S) \text{ and leads to a final state } c \in \mathbb{F} \right)$$

Figure 3.1: Invariant for $\mathrm{A}_{cuts(m)}$ Algorithm (algorithm 2)

Algorithm

We propose here an informal proof of algorithm 2, based on its invariant. Our demonstration follows the schema of proof by invariant presented in [Le Charlier 99], itself based on Hoare's assertion method.

Proof 3.3 (Algorithm "Build $A_{cuts(m)}$")

We have to prove that the following conditions hold.

1. **Initialization** {Inv}
   The initialization phase of the algorithm leads to a state in which all the variables satisfy the invariant.

   It is true because, after the initialization, A_S recognizes only the empty word (it has no transition and its initial state is its final state, too).

   If the language of the chart is not just the empty word, i.e. the chart has more than one cut, the definition of A_Q in the invariant yields that this automaton is not different from $A_{cuts(m)}$ the automaton to build.

2. {Inv $\wedge$ $\mathbb{Q} \neq \emptyset$} **Iteration** {Inv}
   If the invariant and the loop condition are true, then, after one additional iteration in the main loop, the invariant is still true.

   We know, by the invariant, that for every word of the trace set not directly recognized by A_S, A_S is able to recognize a prefix of it and, from the state in which it stopped, A_R can then recognize its suffix. What we do in one iteration is taking one of the initial states in A_R, getting all the events that can occur in this state, i.e. those who do not need any other event to occur before them, and adding these events to the prefix already recognized by A_S. In other words, we take one letter in the suffix of a word and we add it to its prefix.

   Hence, the relationship between A_S and A_Q has not changed. We did not lose any information, even if A_S recognizes some longer prefixes and A_R recognizes some shorter suffixes. The invariant is still true.

3. {Inv $\wedge$ $\mathbb{Q} = \emptyset$ } $\Rightarrow$ {Postcondition}
   When exiting the outer loop, the invariant ensures that the postcondition is fulfilled.

   Since $\mathbb{Q}$ is empty, A_Q cannot recognize anything ($\mathcal{L}(A_Q) = \emptyset$). So, the disjunct Inv_1 in the invariant may not hold and $\mathcal{L}(A_S) = \mathcal{L}_m^{trc}$.

   A symmetric way of getting to the same conclusion is noting that, because $\mathbb{Q}$ is empty, every word recognized by A_S leads automatically to a state in $\mathbb{F}$. And then, using Inv_2 from the invariant, $\mathcal{L}(A_S) = \mathcal{L}_m^{trc}$, which is the postcondition of the algorithm.

□

Knowing that the algorithm is correct, we still have to prove that it will terminate. To do this, we propose the following reasoning.

The induction parameter is the size of A_Q, i.e. its number of states and transitions. It is strictly decreasing because, at each iteration, either a new state is added

to A_S and thus, the number of states in A_Q decreases, or a transition is added between two states in A_S, which takes away a possible transition in A_Q. The size of A_Q has also a lower-bound, namely 0.

Therefore, we know that the algorithm terminates and is correct.

□

Algorithm 2 Build A_{cuts}

Input : The description of an LSC chart m, i.e. its locations, events and $<_m$.

Output : the automaton $< Events(m), \mathbb{S}, \rho, (0, \ldots, 0), \mathbb{F} >$ that recognizes the words belonging to $\mathcal{L}_m^{trc}$ and only those.

{Initialization}
{The initial cut is always discovered}
$\mathbb{S} \leftarrow \{\{0_1, \ldots, 0_n\}\}$
if $\{0_1, \ldots, 0_n\}$ is final **then**
  $\mathbb{F} \leftarrow \{\{0_1, \ldots, 0_n\}\}$
else
  $\mathbb{F} \leftarrow \emptyset$
end if
$\rho \leftarrow \emptyset$

$\mathbb{Q} \leftarrow \{\{0_1, \ldots, 0_n\}\}$

{Iteration}
while $(\mathbb{Q} \neq \emptyset)$ **do**
  {Pick a state in $\mathbb{Q}$ and remove it from $\mathbb{Q}$}
  choose $c \in \mathbb{Q}$
  $\mathbb{Q} \leftarrow \mathbb{Q} \setminus \{c\}$

  {Find all the events that may occur in c}
  $L_\perp \leftarrow \{l \in dom(m) - c \mid \nexists\, \lambda \in dom(m) - c : \lambda <_m l\}$

  for all *location* $\in L_\perp$ **do**
    c' is the *location*-successor of c $\quad (succ_m(c, location, c'))$

    {c' has been discovered}
    $\mathbb{S} \leftarrow \mathbb{S} \cup \{c'\}$
    $\rho \leftarrow \rho \cup \{ < c, e, c' > \}$ where e is the event corresponding to *location*

    {c' is a new state to develop}
    $\mathbb{Q} \leftarrow \mathbb{Q} \cup \{c'\}$

    if c' is final **then**
      $\mathbb{F} \leftarrow \mathbb{F} \cup \{c'\}$
    end if
  end for

end while

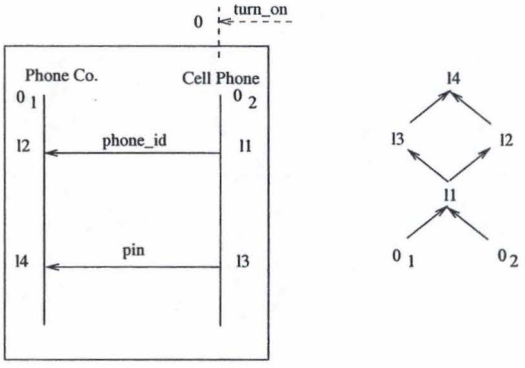


Figure 3.2: An Imaginary Cell Phone Protocol (ICPP)

Example 3.1 (An execution of Build $\mathbf{A}_{cuts(m)}$) *On a small example, we will now show how the automaton recognizing the trace set of a chart is built by our algorithm.*

We use an imaginary cell phone protocol (ICPP), displayed in figure 3.2. In this protocol, the phone sends its identifying number and afterwards sends the personal identification number (pin) of its owner. The communication is asynchronous, hence, the pin can be sent before the identifying number reaches the phone company.

The causal order between location is shown as a Hasse diagram in figure 3.2. There is a path from a location l to a location l', iff, $l <_m l'$.

A_S (the part of $A_{cuts(m)}$ already discovered by the algorithm) is shown in figure 3.3. If the criterion chosen to pick the element of $\mathbb{Q}$ to be developed is First-in, First-out, *this execution is similar to a* breadth-first search *in $A_{cuts(m)}$.*

Below, we write explicitly the values of $L_\perp$ (at the beginning of the iteration) and of $\mathbb{Q}$ (at the end of the iteration). 0_1 and 0_2 represent the initial location in Phone Co. and Cell Phone, respectively.

Step 1

This step is the initialization. $L_\perp$ is not used.

$$\mathbb{Q} = \{\{0_1, 0_2\}\}$$

Step 2

There is only one element in $\mathbb{Q}$, we are thus obliged to develop this one.

$$\begin{aligned} \mathbb{Q} &= \{\{0_1, 0_2, l_1\}\} \\ L_\perp &= \{l_1\} \end{aligned}$$

Step 3

$$\begin{aligned} \mathbb{Q} &= \{\{0_1, 0_2, l_1, l_2\}, \\ &\quad \{0_1, 0_2, l_1, l_3\}\} \\ L_\perp &= \{l_2, l_3\} \end{aligned}$$

Step 4

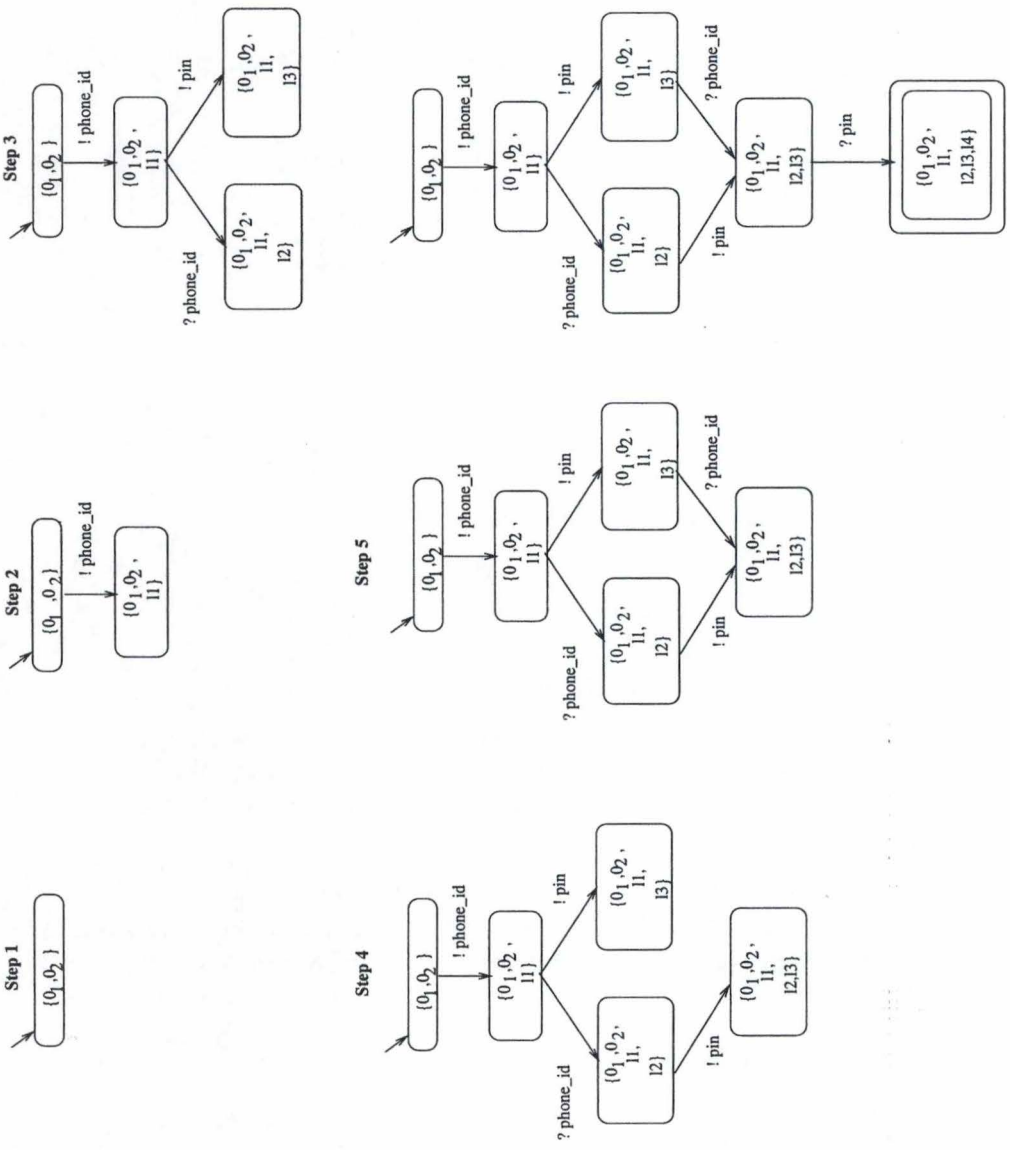


Figure 3.3: How the algorithm 2 discovers the automaton for ICPP.

In this step, we choose to develop the cut $\{0_1, 0_2, l_1, l_2\}$.

$$\begin{aligned} \mathbb{Q} &= \{\{0_1, 0_2, l_1, l_2, l_3\}, \\ &\quad \{0_1, 0_2, l_1, l_3\}\} \\ L_\perp &= \{l_3\} \end{aligned}$$

Step 5

Now, according to our FIFO strategy, we develop $\{0_1, 0_2, l_1, l_3\}$. *This development does not add any new state to* $\mathbb{Q}$.

$$\begin{aligned} \mathbb{Q} &= \{\{0_1, 0_2, l_1, l_2, l_3\}\} \\ L_\perp &= \{l_2\} \end{aligned}$$

Step 6

$$\begin{aligned} \mathbb{Q} &= \{\{0_1, 0_2, l_1, l_2, l_3, l_4\}\} \\ L_\perp &= \{l_4\} \end{aligned}$$

Step 7

This additional step just removes the last cut from $\mathbb{Q}$.

$$\begin{aligned} \mathbb{Q} &= \emptyset \\ L_\perp &= \emptyset \end{aligned}$$

3.2.3 Assemble the main chart and its prechart

Using the solutions to the first two subproblems, we are able to build an automaton $A_{prech(m)}$ recognizing $\mathcal{L}^{trc}_{prech(m)}$ and an automaton A_m recognizing $\mathcal{L}^{trc}_m$.

Once we have these automata, we just add an ε-transition from every final state in $A_{prech(m)}$ to the initial state of A_m.

We can then determinize this automaton, if it is needed for the remaining steps of the algorithm. For the basic method presented now, it is not necessary. For the optimized methods presented later, see chapter 4, we require this automaton to be in minimal deterministic form. There are well-known algorithms to perform this task, see [Aho 00, Wilhelm 94].

3.2.4 Generate the formula from the trace set

The trace sets being finite, we can walk through the automata representing them to enumerate their words and generate the formula on-the-fly. Automata are a good representation of finite languages, because they are compact while allowing an easy manipulation.

Moreover, as they can be considered as directed graphs, algorithms from graph theory can be used on them.

3.3 Complexity issues

We will first study the time complexity of the translation problem and then, we will analyze the complexity of the particular algorithm that we propose in this report.

3.3.1 Problem

We consider the time complexity of translating a chart m into its equivalent formula φ_m, where φ_m has the form given in definition 3.3.

We call the problem addressed in this report **small LSC2CTL*** and we define it as:

Definition 3.6 (small LSC2CTL*) *Given a chart m belonging to the subset of LSC described in this document, build its equivalent CTL* formula φ_m, where φ_m respects definition 3.3*

It is noteworthy that we do not analyze the complexity of the *general* **LSC2CTL*** problem, that we define as:

Definition 3.7 (LSC2CTL*) *Given a chart m belonging to the subset of LSC described in this document, build a CTL* formula ϕ_m such that:*

$$\forall \mathcal{I} \; : \; \mathcal{I} \models m \iff \mathcal{I} \models \phi_m$$

We take the amount of locations in a chart m as a measure of its size. Hence, $|dom(m)|$ will represent the size of the problem.

The **small LSC2CTL*** translation problem requires at least to enumerate every word in the trace set of the chart, in order to generate the formula. Therefore, it will take up to $|\mathcal{L}| \times |m|$ computation steps to enumerate the words of a language $\mathcal{L}$, where $|\mathcal{L}|$ denotes the number of words in the language considered and $|m|$ represents the size of a word in $\mathcal{L}$.

Here, $\mathcal{L} = \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m}$ and the maximum size of a word is $|dom(prech(m))| + |dom(m)|$, the number of locations in the prechart plus the number of locations in the main chart.

Independently from the representation of the words chosen, we can thus say that the problem is in $time(O(|\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m}|(|dom(prech(m))| + |dom(m)|)))$ where $|\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m}|$ is the number of words belonging to the trace set of the chart concatenated to the trace set of the prechart. Since the trace sets consist of permutations of events, when appended, their size is $O(|dom(prech(m))|! \; |dom(m)|!) = O(|dom(prech(m)| + |dom(m)|^{|dom(prech(m)|+|dom(m)|})$.

Theorem 3.3 (Translation complexity is more than exponential) *The* **small LSC2CTL*** *problem is in time $O(n\, n^n)$ where $n = |dom(prech(m))| + |dom(m)|$.*

It would be interesting to study the time complexity of **LSC2CTL***, since this could yield some information about the succinctness of LSC with regards to CTL*. Regarding the expressiveness of LSC, it is already known that the subset of LSCs considered here is less expressive than CTL* [Harel 00b].

3.3.2 Algorithm

Build $<_m$

This step must build a binary relation on $dom(m)$, thus $<_m \subseteq dom(m) \times dom(m)$.

Hence, its time complexity is $O(|dom(m)|^2)$.

The space complexity is the size of an acyclic graph representing the relation, i.e. $O(|dom(m)|)$.

Build $A_{cuts(m)}$

As stated in example 3.1, if the criterion to choose the state to develop is FIFO, the execution is similar to a Breadth-First Search (BFS) in $A_{cuts(m)}$. Hence, its time complexity is linear in the size of this automaton.

$|A_{cuts(m)}| = |S| + |\rho|$ where S denotes the set of states in $A_{cuts(m)}$ and ρ the set of transitions. Since ρ is a binary relation on S, we deduce that $|\rho| \leq |S|^2$.

Referring to definition 3.1, we know that $S = cuts(m) \subseteq 2^{dom(m)}$. Hence, $|S| = |cuts(m)| \leq 2^{|dom(m)|}$ and $|A_{cuts(m)}| = O(2^{|dom(m)|} + 2^{2|dom(m)|}) = O(2^{|dom(m)|})$

From these facts, we deduce that our solution to this subproblem is in time $O(2^{|dom(m)|})$.

The algorithm uses the work variable $\mathbb{Q} \subseteq cuts(m) \subseteq 2^{dom(m)}$. Consequently, its space complexity is $O(2^{|dom(m)|})$.

Append the trace set of the chart and the prechart

The time complexity of this step is linear in the number of final states of the prechart.

The number of states in an automaton representing a chart m being $O(2^{|dom(m)|})$, we deduce that the time complexity of this step is $O(2^{|dom(prech(m)|})$.

Its space complexity is of no importance, because this step does not use any work space.

Generate the formula

Since we have to enumerate every word in the trace set, this step achieves a time complexity of $O((|dom(prech(m))|!\ |dom(m)|!)(|dom(prech(m))| + |dom(m)|)) = O(n\, n^n)$, where $n = |dom(prech(m))| + |dom(m)|$.

The space complexity is of no importance because it does not use any work space, either.

For this step has the highest time complexity, it constraints the time complexity of the whole algorithm, while the space complexity is determined by the second step, *Build* $A_{cuts(m)}$. This result is summarized in the following theorem.

Theorem 3.4 (Time and space complexity of our algorithm) *Algorithm 2, solving the* **small LSC2CTL*** *translation problem is in*

1. *time* $O(n\, n^n)$
2. *space* $O(2^n)$ *(thus, in* **EXPSPACE***)*

where $n = |dom(prech(m))| + |dom(m)|$.

Chapter 4

Transforming Charts Formulae

4.1 Optimizing charts formulae for model-checking

At the time being, model-checking is still a hard problem to solve. It requires much time and space and it is not unusual to encounter a time complexity exponential in the size of the formulae[1]. If we want to check real-world scenarios, involving multiple instances, long protocols and, moreover, inducing many possible traces, we will have to cope with this performance issue, in order to keep the verification practically feasible.

In this section, we aim at transforming the general formulae presented above (p.38) into several smaller formulae. The reason to do so is that it is preferable to do twice something taking one hour than to do it all at once, if it takes ten hours. Or, returning the exponential complexity issue, if the size of a problem can be reduced, the time needed to solve it will decrease more than proportionally. However, we must be very careful when performing such transformations, since the *split* formulae obtained *must* be equivalent to the original *non-split* formula.

4.1.1 Existential charts

We note that the formula

$$\varphi_m = EF\Big(\bigvee_{w \in \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m} \phi_w\Big)$$

is equivalent to

$$\varphi_m = \bigvee_{w \in \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m} EF(\phi_w)$$

This means that we will only need to check that there exists a path in which one word of the trace set finally appears.

We will not necessarily need to check all the subformulae. We can do some kind of "lazy evaluation", stopping it as soon as one word of the trace set is matched by the implementation.

4.1.2 Universal charts

Let us first present the intuition behind the splitting of a universal formula.

It is obvious that we cannot use the same method as for an existential chart. Indeed, simply moving the $\vee$ operators from *inside* the scope of the AG operator to *outside* would result in non-equivalent formulae, because
$AG(a \vee b) \not\Leftrightarrow AG(a) \vee AG(b)$.

Our idea is to find a special event around which the evaluation of the formula could be divided into two parts. We would like this event to uniquely identify the state in which it appears. So, we could prove the formula until we reach the state identified by this event and, afterwards, start again the proof of the formula from this state. Let us illustrate this concept on a short example.

[1] Model-checking of LTL and CTL* formulae are both exponential in the size of the formula while CTL model-checking is linear in size of the formula, see [Dams 96].

Example 4.1 *Consider a universal chart activated by a, the trace of which is* $\{e_1 \cdot e_2 \cdot e_3\}$. *Remember, from the general definition of a formula, that N represents* $\neg(e_1 \vee e_2 \vee e_3)$. *Using the general formula of definition 3.4, the CTL* formula equivalent to the chart is*

$$AG(a \rightarrow a \wedge X(N\ U(\ e_1 \wedge X(N\ U(\ e_2 \wedge X(N\ U\ e_3\ ))))))$$

A sequence of states satisfying this formula, when the activation event a occurs, would have the following form :

$$\underbrace{\sigma_0}_{\models a}\ \overbrace{\cdots\cdots}^{\models N}\ \underbrace{\sigma_1}_{\models e_1}\ \overbrace{\cdots\cdots}^{\models N}\ \underbrace{\sigma_2}_{\models e_2}\ \overbrace{\cdots\cdots}^{\models N}\ \underbrace{\sigma_3}_{\models e_3}$$

We could split this sequence into two parts, around σ_2, *since we think that* σ_2 *is uniquely identified by* e_2. *So, we would have the following two sequences*

$$(1)\quad \underbrace{\sigma_0}_{\models a}\ \overbrace{\cdots\cdots}^{\models N}\ \underbrace{\sigma_1}_{\models e_1}\ \overbrace{\cdots\cdots}^{\models N}\ \underbrace{\sigma_2}_{\models e_2}$$

$$(2)\quad \underbrace{\sigma_0}_{\models a}\ \overbrace{\cdots\cdots\cdots\cdots\cdots}^{\models \neg e_2}\ \underbrace{\sigma_2}_{\models e_2}\ \overbrace{\cdots\cdots}^{\models N}\ \underbrace{\sigma_3}_{\models e_3}$$

that are models, respectively, for the following formulae :

$$(1)\quad AG(a \rightarrow a \wedge X(N\ U(\ e_1 \wedge X(N\ U(\ e_2)))))$$
$$(2)\quad AG(a \rightarrow (\neg e_2)U(\ e_2 \wedge X(N\ U(\ e_3))))$$

The most intricate issue with this splitting is undoubtedly finding the events in the formula that are able to uniquely identify a state appearing after the state satisfying the activation event. This state is critical since the evaluation of the *head* of the formula will end on it and the evaluation of the *tail* of the formula will start again from it.

We now present the idea behind the characterization of this event, for the general multi-trace chart.

Every chart has a trace set. This trace set, as already emphasized, is a regular language, thus recognized by an ADFA (Acyclic Deterministic Finite Automaton). Let us assume that this automaton is minimal and can be divided into three parts, as shown on figure 4.1:

- a set of states *before* an edge labelled by p, containing no final state
- an edge labelled by p
- a set of states *after* this edge, containing at least one accepting state.

We want this edge to fulfill some particular properties, namely that *every path recognizing a word of the trace set has to use it* and that there is no edge labelled with p in the *before*-part of the automaton.

If it is the case, we can then split the recognition of every word of the trace set into two parts : the first one being composed of all the letters from the beginning of the word to p, including p, and the second one, starting again from p, to the end of the word.

Now, we will define more formally and generally this special edge and show how it can be used to split up a universal formula. [2]

[2]For the sake of simplicity, we spoke of *one* edge here. It is actually a particular case of the general characterization presented later (proposition 4.3 on page 68).

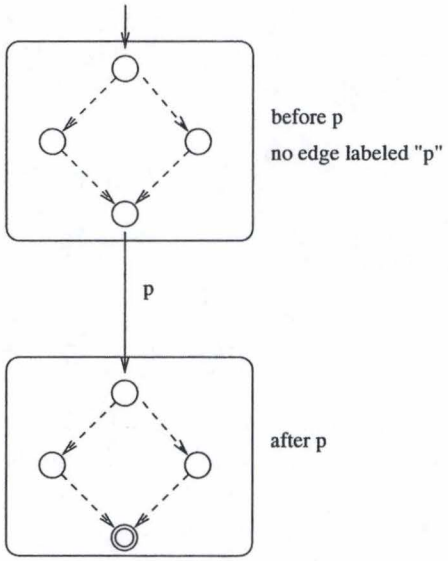


Figure 4.1: A minimal DFA having a pivot p

Pivots

In appendix A, we give a generalized and formal definition of the concepts of pivot and formula splitting, by use of pivoting. In this section, we only present a particular case of this more general theory but we believe that it gives a good intuition of what this concept represents.

As usual, we take Σ as a finite alphabet. We say that $\mathcal{L} \subseteq \Sigma^*$ is a *language over* Σ if $\mathcal{L}$ is a set containing finite sequences of characters (or *letters*) belonging to Σ. We require that $\mathcal{L}$ is finite, i.e. $|\mathcal{L}| < \infty$.

As a preliminary notion, let us define two useful functions, returning respectively the *letters* of a word and a language. Since they are unambiguously identified by their signature, we give them the same name :

Definition 4.1 (Letters)

$$\begin{array}{lcl} letters(w) & = & \{\, a \mid a \in \Sigma \setminus \{\varepsilon\} \wedge \exists\, w', w'' \,:\, w = w' \cdot a \cdot w'' \} \\ letters(\mathcal{L}) & = & \{\, a \mid \exists\, w \in \mathcal{L} \,:\, a \in letters(w) \} \end{array}$$

A pivot in a language is a letter allowing us to divide every word w into two words w_1 and w_2, such that we can decide if w belongs to the given language by looking at w_1 and w_2, indepently.

Definition 4.2 (Pivot) *The letter* p *is defined as a* **pivot in a language** $\mathcal{L}$ *iff it satisfies the following conditions* [3] *:*

1.

$$p \neq \varepsilon$$

2.

$$\forall\, w \in \mathcal{L} \,:\, \exists!\, w_1, w_2 \,:\, w = w_1 \cdot p \cdot w_2 \wedge p \notin letters(w_1)$$

[3] This definition is not minimal, since condition (1) is deduced from the fact that p is a letter.

3.

$$\begin{array}{lll} \forall\ y_1 \cdot p \cdot y_2 \in \mathcal{L} & , & w_1 \cdot p \cdot w_2 \in \mathcal{L} : \\ p \notin \mathit{letters}(y_1) & \wedge & p \notin \mathit{letters}(w_1) \\ & \Downarrow & \\ w_1 \cdot p \cdot y_2 \in \mathcal{L} & \wedge & y_1 \cdot p \cdot w_2 \in \mathcal{L} \end{array}$$

The second condition allows us to define two important notions,

Definition 4.3 (Pivot-prefix and Pivot-suffix) *Referring to condition (2) of definition 4.2, w_1 is the* pivot-prefix *of w by p, written pivot-prefix(w,p) and w_2 is the* pivot-suffix *of w by p, written pivot-suffix(w,p).*

Note that, for a given word and a pivot, the pivot-prefix and pivot-suffix are uniquely defined. Their emptiness is possible.

The last condition tells us that every combination of *pivot-prefixes* and *pivot-suffixes* around a pivot produces a word of the language considered. This means, in particular, that a language in which all the words start with the same letter a has a as a pivot.

We extend the notion of *pivot-prefix* (sym. *pivot-suffix*) to a language.

Definition 4.4 (Pivot-prefixes of a language) *We say that $\mathcal{L}'$ is a* pivot-prefix *of a language $\mathcal{L}$ by* p*, written $\mathcal{L}' = \mathit{pivot\text{-}prefix\text{-}lang}(\mathcal{L},p)$ iff*

1. p *is a pivot in $\mathcal{L}$*
2. $\mathcal{L}' = \{\ w \mid \exists\ w' \in \mathcal{L} :\ w = \mathit{pivot\text{-}prefix}(w',p)\}$

A given language can have zero, one or more pivots. If it has many pivots, we can then define an order between them, with respect to the order in which they appear in every word of the language. By the definition of a pivot and the fact that we deal with *finite languages*, it is clear that this order is *always* the same, regardless the particular word taken. We prove this claim later on.

Proposition 4.1 (Pivots appear always in the same order) *Let $\mathcal{L}$ be a finite language, having two pivots p,p'. Then, we have that*

$$\forall w \in \mathcal{L}\ :\ \exists w_1,w_2,w_3\ :\ w = w_1 \cdot p \cdot w_2 \cdot p' \cdot w_3$$

(with $w_1 = \mathit{pivot\text{-}prefix}(w,p)$ and $w_3 = \mathit{pivot\text{-}suffix}(w,p')$) or

$$\forall w \in \mathcal{L}\ :\ \exists w_1,w_2,w_3\ :\ w = w_1 \cdot p' \cdot w_2 \cdot p \cdot w_3$$

(with $w_1 = \mathit{pivot\text{-}prefix}(w,p')$ and $w_3 = \mathit{pivot\text{-}suffix}(w,p)$)

Proof 4.1 (Proposition 4.1) *We have that $\mathcal{L}$ is a finite language with two pivots p,p'. For the sake of contradiction, we assume that there are two words $w,w' \in \mathcal{L}$ such that*

$$\begin{array}{rcl} w & = & w_1 \cdot p \cdot w_2 \cdot p' \cdot w_3 \\ w' & = & w'_1 \cdot p' \cdot w'_2 \cdot p \cdot w'_3 \end{array}$$

with

$$\begin{aligned} w_1 &= \textit{pivot-prefix}(w,p) \\ w_3 &= \textit{pivot-suffix}(w,p') \\ w'_1 &= \textit{pivot-prefix}(w',p') \\ w'_3 &= \textit{pivot-suffix}(w',p) \end{aligned}$$

and, thus,

$$\begin{aligned} w_1 \cdot p \cdot w_2 &= \textit{pivot-prefix}(w,p') \\ w'_1 \cdot p' \cdot w'_2 &= \textit{pivot-prefix}(w',p) \end{aligned}$$

Hence, by the fact that p' is a pivot, we have, as condition (3) of definition 4.2 states, that $w^1 = w_1 \cdot p \cdot w_2 \cdot p' \cdot w'_2 \cdot p \cdot w'_3$ is also a word of $\mathcal{L}$.

We have that pivot-prefix$(w^1,p) = w_1$. This yields that $w^2 = w'_1 \cdot p' \cdot w'_2 \cdot p \cdot w_2 \cdot p' \cdot w'_2 \cdot p \cdot w'_3$ is in $\mathcal{L}$, too, with pivot-prefix$(w^2,p') = w'_1$.

We can use the same argument (switching the pivot-prefix of the last w^i with a longer pivot-prefix of w or w') as often as we want. Thus, this yield that, necessarily, $\mathcal{L}$ contains an infinite number of words, which is impossible, since we assumed that $\mathcal{L}$ was finite.

Thus, we should conclude that either p must always appear before p' or after p'.

□

Definition 4.5 (Precedence order) *If p and p' are pivots in $\mathcal{L}$, we will say that p* precedes *p', written $p \prec p'$, if* every *word in pivot-prefix-lang($\mathcal{L},p$) is a* **strict prefix** *of* at least one *word of pivot-prefix-lang($\mathcal{L},p'$) or, dually,* every *word in pivot-suffix-lang($\mathcal{L}$,p') is a* **strict suffix** *of* at least one *word of pivot-suffix-lang($\mathcal{L}$,p).*

The following proposition is central for the method presented later.

Proposition 4.2

$$p\ ,\ p' \textit{ are pivots in } \mathcal{L} \textit{ with } p \prec p'$$

$$\Downarrow$$

$$p' \textit{ is a pivot in } \mathcal{L}' = \Big(\{p\} \cdot \textit{pivot-suffix-lang}(\mathcal{L},p)\Big)$$

To understand the intuition behind it, refer to the automaton of figure 4.2. We see that removing the set of states A from this automaton will not change the status of p' as a pivot. Below is a demonstration of this proposition, based on the formal definitions presented above.

Proof 4.2 *We have to show that* p' *fulfills the three conditions defining a pivot, for the language $\mathcal{L}'$.*

It is trivial that the first two conditions hold. We demonstrate that the third holds, too.

Let us take two words in $\mathcal{L}'$. They have the form $p \cdot w_1 \cdot p' \cdot y_1$ and $p \cdot w_2 \cdot p' \cdot y_2$.

We have to prove that $p \cdot w_1 \cdot p' \cdot y_2$ and $p \cdot w_2 \cdot p' \cdot y_1$ are both in $\mathcal{L}'$.

Note that, since $w_1 \cdot p' \cdot y_1$ and $w_2 \cdot p' \cdot y_2$ are pivot-suffixes *of p in $\mathcal{L}$ and p is a pivot, there are necessarily x_1 and x_2 in pivot-prefix-lang($\mathcal{L},p$) such that $x_1 \cdot p \cdot w_1 \cdot p' \cdot y_1$ and $x_2 \cdot p \cdot w_2 \cdot p' \cdot y_2$ are in $\mathcal{L}$.*

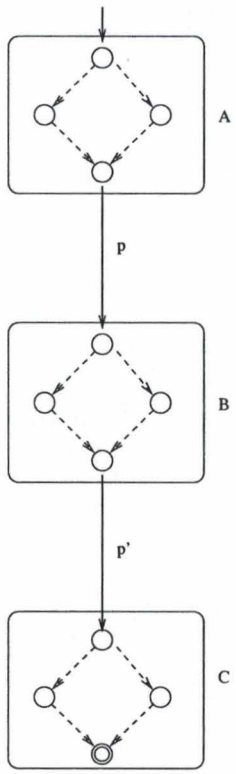


Figure 4.2: Illustration of proposition 4.2

Because p' is a pivot in $\mathcal{L}$, we know, by the third condition of definition 4.2, that $x_1 \cdot p \cdot w_1 \cdot p' \cdot y_2$ is also a word of $\mathcal{L}$. Hence, $w_1 \cdot p' \cdot y_2$ is a pivot-suffix *of p in $\mathcal{L}$ and $p \cdot w_1 \cdot p' \cdot y_2$ is in $\mathcal{L}'$.*

The second part of the demonstration is as straightforward as the first. We know that p' is a pivot in $\mathcal{L}$, yielding that $x_2 \cdot p \cdot w_2 \cdot p' \cdot y_1$ is in $\mathcal{L}$.

Then, it comes that $w_2 \cdot p' \cdot y_1$ is a pivot-suffix *of p in $\mathcal{L}$. Therefore, $p \cdot w_2 \cdot p' \cdot y_1$ is in $\mathcal{L}'$.*

□

Our ultimate goal is, given the trace set of a chart, to divide this language into languages containing shorter words. Then, we could reuse the formula φ_m, that was quantified over $\mathcal{L}_m^{trc}$, over the languages obtained from the division. Hence, we would have several formulae that, when conjuncted, are equivalent to the φ_m formula. We use proposition 4.2 above to perform this division, as the following definition states.

Definition 4.6 (Division of a language by its pivots) *Let $\mathcal{L}$ be a finite language, having $\{p_1, \ldots, p_k\}$ pivots, with $p_1 \prec \ldots \prec p_k$. Then, the* division *of $\mathcal{L}$ by $\{p_1, \ldots, p_k\}$, written $\mathcal{L}|_{\{p_1,\ldots,p_k\}}$ is defined as:*

$$\mathcal{L}|_{\{p_1,\ldots,p_k\}} = \mathcal{L}_1, \ldots, \mathcal{L}_{k+1}$$

where

$$\begin{aligned}
\mathcal{L}_1 &= \textit{pivot-prefix-lang}(\mathcal{L}, p_1) \\
\mathcal{L}_2 &= \textit{pivot-prefix-lang}(\{p_1\} \cdot \textit{pivot-suffix-lang}(\mathcal{L}, p_1), p_2) \\
&\vdots \\
\mathcal{L}_i &= \textit{pivot-prefix-lang}(\{p_{i-1}\} \cdot \textit{pivot-suffix-lang}(\mathcal{L}, p_{i-1}), p_i) \\
&\vdots \\
\mathcal{L}_{k+1} &= \{p_k\} \cdot \textit{pivot-suffix-lang}(\mathcal{L}, p_k)
\end{aligned}$$

Note that this operation is invertible,

Property 4.1 (Division is invertible)

$$\mathcal{L} = \mathcal{L}_1 \cdot \mathcal{L}_2 \ldots \cdot \mathcal{L}_i \cdot \ldots \cdot \mathcal{L}_{k+1}$$

and that, for a pivot p_i,

Property 4.2

$$\textit{pivot-prefix-lang}(\mathcal{L}, p_i) = \mathcal{L}_1 \cdot \ldots \cdot \mathcal{L}_i$$

Proof 4.3 (Division is invertible) *Since $\mathcal{L}_1 \cdot \ldots \cdot \mathcal{L}_k = \textit{pivot-prefix-lang}(\mathcal{L}, p_k)$, as property 4.2 states, and $\mathcal{L}_{k+1} = \{p_k\} \cdot \textit{pivot-suffix-lang}(\mathcal{L}, p_k)$, it comes that $\mathcal{L} = \textit{pivot-prefix-lang}(\mathcal{L}, p_k) \cdot \{p_k\} \cdot \textit{pivot-suffix-lang}(\mathcal{L}, p_k)$, i.e. $\mathcal{L} = \mathcal{L}_1 \cdot \ldots \mathcal{L}_{k+1}$.*

□

This last proof relied on property 4.2. Here comes the proof of this property, by recursion on the number of pivots.

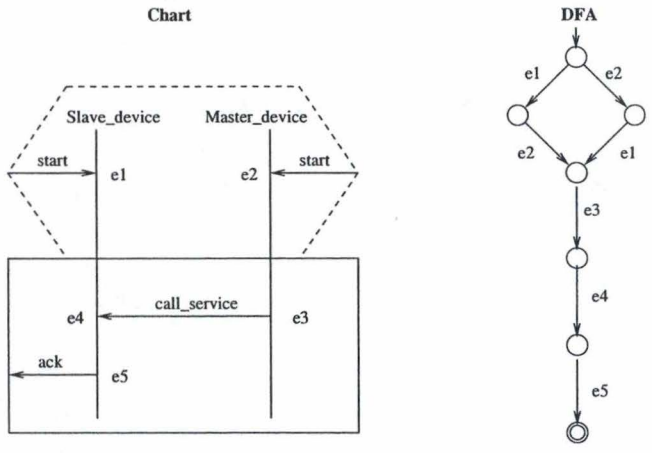


Figure 4.3: A chart and the DFA recognizing its trace set

Proof 4.4 (*pivot-prefix-lang*$(\mathcal{L}, p_i) = \mathcal{L}_1 \cdot \ldots \cdot \mathcal{L}_i$) *The proof is done by recursion on the number of pivots used in the division.*

The base is $i = 1$. *By definition, we have pivot-prefix-lang*$(\mathcal{L}, p_1) = \mathcal{L}_1$.

Now we should prove that for i *such that* $1 < i \leq k$, $\mathcal{L}_1 \cdot \ldots \cdot \mathcal{L}_{i-1} \cdot \mathcal{L}_i =$ *pivot-prefix-lang*$(\mathcal{L}, i)$. *Induction hypothesis yields that* $\mathcal{L}_1 \cdot \ldots \cdot \mathcal{L}_{i-1} =$ *pivot-prefix-lang*$(\mathcal{L}, i-1)$. *Referring to the definition of* $\mathcal{L}_i$, *we may say that* $\mathcal{L}_i =$ *pivot-prefix-lang*$(\{p_{i-1}\} \cdot$ *pivot-suffix-lang*$(\mathcal{L}, p_{i-1}), p_i)$. *We now decompose the right-hand term.*

First of all, every word in $\{p_{i-1}\} \cdot$ *pivot-suffix-lang*$(\mathcal{L}, p_{i-1})$ *has the form* $p_{i-1} \cdot w_1 \cdot p_i \cdot w_2$, *with* $p \notin letters(w_1)$.

Secondly, the pivot-prefixes of these words have the following form: $p_{i-1} \cdot w_1$.

Now, it is obvious that, if w_0 *is a word in pivot-prefix-lang*$(\mathcal{L}, p_{i-1})$, *then* $w_0 \cdot p_{i-1} \cdot w_1$ *is a word of pivot-prefix-lang*$(\mathcal{L}, p_i)$.

□

In order to clarify these concepts, here is a short example.

Example 4.2 *On figure 4.3, we present a chart describing a protocol in which two machines, when switched on by the user, interact and then confirm that their work ended successfully. The automaton recognizing the trace set of this chart is also displayed in figure 4.3. The trace set can be enumerated as follows :*

$$\mathcal{L} = \{ \; e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_5, \; e_2 \cdot e_1 \cdot e_3 \cdot e_4 \cdot e_5 \}$$

The pivots of $\mathcal{L}$ *are* e_3, e_4 *and* e_5

$$\begin{aligned} \textit{pivot-prefix-lang}(\mathcal{L}, e_3) &= \{e_1 \cdot e_2 \; , \; e_2 \cdot e_1\} \\ \textit{pivot-suffix-lang}(\mathcal{L}, e_3) &= \{e_4 \cdot e_5\} \end{aligned}$$

$$\begin{aligned} \textit{pivot-prefix-lang}(\mathcal{L}, e_4) &= \{e_1 \cdot e_2 \cdot e_3 \; , \; e_2 \cdot e_1 \cdot e_3\} \\ \textit{pivot-suffix-lang}(\mathcal{L}, e_4) &= \{e_5\} \end{aligned}$$

$$\begin{aligned} \textit{pivot-prefix-lang}(\mathcal{L}, e_5) &= \{e_1 \cdot e_2 \cdot e_3 \cdot e_4 \; , \; e_2 \cdot e_1 \cdot e_3 \cdot e_4\} \\ \textit{pivot-suffix-lang}(\mathcal{L}, e_5) &= \{\varepsilon\} \end{aligned}$$

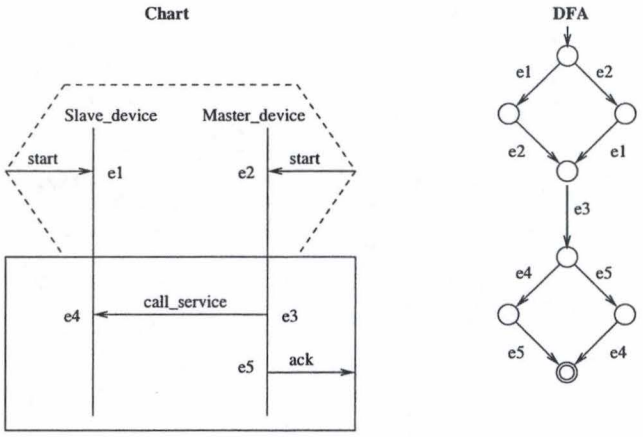


Figure 4.4: The chart of example 4.2 revisited

We have $e_3 \prec e_4 \prec e_5$, *as expected.*

We can also see that the proposition 4.2 is true, since both e_4 *and* e_5 *are pivots in* $\{e_3\} \cdot \{e_4 \cdot e_5\}$ *and* e_5 *is a pivot in* $\{e_4\} \cdot \{e_5\}$.

The division of $\mathcal{L}$ *by its pivots gives the following languages :*

$$\begin{aligned} \mathcal{L}|_{\{e_3,e_4,e_5\}} &= \{e_1 \cdot e_2, e_2 \cdot e_1\} = \mathcal{L}_1, \\ &\quad \{e_3\} = \mathcal{L}_2, \\ &\quad \{e_4\} = \mathcal{L}_3 \\ &\quad \{e_5\} = \mathcal{L}_4 \end{aligned}$$

Why is not e_2 *a pivot ? Because considering it as a pivot would violate condition* (3). *Let us assume that* $w_1 = e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_2 \cdot e_5$ *and* $w_2 = e_2 \cdot e_1 \cdot e_3 \cdot e_4 \cdot e_2 \cdot e_5$. *We would have pivot-prefix*$(w_1, e_2) = e_1$ *and thus, as one can immediately see,* $e_1 \cdot w_2 \notin \mathcal{L}$.

The reader should be aware that this definition is more restrictive than it may seem at first sight. Let us slightly modify the language of the chart of figure 4.3, used in example 4.2.

The new chart and its DFA are shown on figure 4.4.

Now, it is no longer the slave machine that confirms the successful termination of the work but the master machine, as soon as it has requested the service from the slave. Note that the meaning of this protocol is quite different from the previous one. Here, the service request does not have to be effectively received before the user is acknowledged. However, we are sure that this query will still be received by the slave. Indeed, we made the assumption that the communication was reliable.

Example 4.3

$$\begin{aligned} \mathcal{L}' = \{ & e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_5, \\ & e_1 \cdot e_2 \cdot e_3 \cdot e_5 \cdot e_4, \\ & e_2 \cdot e_1 \cdot e_3 \cdot e_4 \cdot e_5 \\ & e_2 \cdot e_1 \cdot e_3 \cdot e_5 \cdot e_4\} \end{aligned}$$

e_4 and e_5 are no longer pivots in it. Because, if they were, they would violate condition (3). Actually, we see that every combination of *pivot-prefixes* and *pivot-suffixes* around e_4 does not produce a word of $\mathcal{L}'$ anymore. For example, $e_1 \cdot e_2 \cdot e_3 \cdot e_5 \cdot e_4 \cdot e_5$ is not a word of $\mathcal{L}'$. The same method can be applied to e_5 to show that it is not a pivot.

Splitting up the formula (Method 1)

Remember from definition 3.4 that the general formula for a universal chart is

$$\varphi_m = \mathrm{AG}\Big(\bigvee_{p \in \mathcal{L}^{trc}_{prech(m)}} \phi_p \rightarrow \bigvee_{w \in \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m} \phi_w\Big)$$

where

$$\phi_{q_1 \cdot \ldots \cdot q_n} = q_1 \wedge X(N U (q_2 \wedge \ldots X(N U q_n)) \ldots)$$

Let $p_1, \ldots, p_k$ be pivots in $\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m$ satisfying the following requirements:

1. *pivot-prefix-lang*$(\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m, p_1) \neq \{\varepsilon\}$
2. *pivot-suffix-lang*$(\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m, p_k) \neq \{\varepsilon\}$
3. $p_1 \prec \ldots \prec p_k$

Let $(\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m)|_{\{p_1,\ldots,p_k\}} = \mathcal{L}_1, \ldots, \mathcal{L}_{k+1}$.

The split formulae are then

Definition 4.7 (Split universal formulae)

$$\begin{aligned}
\varphi_0 &= AG\Big(\bigvee_{p \in \mathcal{L}^{trc}_{prech(m)}} \phi_p \rightarrow \bigvee_{w \in \mathcal{L}_1 \cdot \{p_1\}} \phi_w\Big) \\
\varphi_1 &= AG\Big(\bigvee_{p \in \mathcal{L}^{trc}_{prech(m)}} \phi_p \rightarrow \bigvee_{w \in \mathcal{L}_2 \cdot \{p_2\}} \vartheta_w\Big) \\
&\vdots \\
\varphi_k &= AG\Big(\bigvee_{p \in \mathcal{L}^{trc}_{prech(m)}} \phi_p \rightarrow \bigvee_{w \in \mathcal{L}_{k+1}} \vartheta_w\Big)
\end{aligned}$$

where,

$$\begin{aligned}
\phi_{q_1 \cdot \ldots \cdot q_n} &= q_1 \wedge X(N U \ldots \wedge X(N U\ q_n) \ldots) \\
\vartheta_{q_1 \cdot \ldots \cdot q_n} &= (\neg q_1)\ U (q_1 \wedge X(N U \ldots \wedge X(N U\ q_n) \ldots) \\
N &= \bigwedge_{e \in Events(m)} \neg e
\end{aligned}$$

A formula φ_i states that, whenever the prechart holds, we have to retrieve the first occurrence of the $(i-1)$-th pivot $((\neg p_{i-1})\ U\ p_{i-1})$ and then, we check that all the traces contained in the fragment of $\mathcal{L}^{trc}_m$ between p_{i-1} and p_i is satisfied by the chart. $\mathcal{L}_i$ represents the *fragment* of $\mathcal{L}^{trc}_m$ contained between p_{i-1} and p_i.

Remember, from the definition of a division of a language by its pivots (definition 4.6), that the first letter of every word in the language $\mathcal{L}_{i+1}$ is *always* the pivot p_i. Moreover, p_i is also the last letter of every word in the language $\mathcal{L}_i \cdot \{p_i\}$. The state identified by p_i is thus, at the same time, the state on which the evaluation of the formula φ_{i-1} ended and the state on which the evaluation of φ_i will begin.

Intuitively, if we are sure that the last state of φ_{i-1} is always the same state as the state of φ_i, we will know that the split formulae express precisely the same property as the general unique formula, φ_m. The proof of this equivalence is done below and is based on the intuitive idea that no confusion could be introduced

between states in which p_i holds, since p_i is a pivot. Hence, no part of the run could be *forgotten* or evaluated twice, thus the runs have been correctly divided.

The following equivalence should now be proved :

Theorem 4.1 (Equivalence of split and non-split formulae)

$$\forall\, \mathcal{I} : \mathcal{I} \models \bigwedge_{j=0}^{k} \varphi_j \quad \Longleftrightarrow \quad \mathcal{I} \models \varphi_m$$

Proof 4.5 (Equivalence 4.1) *We demonstrate the necessary and sufficient conditions separately.*

The necessary condition is proven by induction on k, the number of pivots.
Remember that $\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m} = \mathcal{L}_1 \cdot \ldots \cdot \mathcal{L}_{k+1}$.
We have to show that, for any KS $\mathcal{I}$

$$\mathcal{I} \models \bigwedge_{j=0}^{k} \varphi_j \quad \Rightarrow \quad \mathcal{I} \models \varphi$$

We suppose that $\mathcal{I} \models \bigwedge_{j=0}^{k} \varphi_j$ *and we want to show that* $\mathcal{I} \models \varphi$.

Regarding $\mathcal{I} \models \bigwedge_{j=0}^{k} \varphi_j$, *there can be two situations :*

1. $\mathcal{I} \models AG(\bigwedge_{p \in \mathcal{L}^{trc}_{prech(m)}} \neg \phi_p)$

   In which case,trivially, $\mathcal{I} \models \varphi$.

2. $\mathcal{I} \models EF(\phi_p)$, *where* $p \in \mathcal{L}^{trc}_{prech(m)}$.

   This means that one of the words of the prechart appears at least once. Let us develop this non-trivial case, using the induction hypothesis[4]

$$\forall i : 0 \leq i < k : \mathcal{I} \models \bigwedge_{j=0}^{i} \varphi_j \Rightarrow \mathcal{I} \models AG(\bigvee_{p \in \mathcal{L}^{trc}_{prech(m)}} \phi_p \rightarrow \bigvee_{w \in \mathcal{L}_1 \cdot \ldots \cdot \mathcal{L}_{i+1} \cdot \{p_{i+1}\}} \phi_w)$$

We know that both $EF(\phi_p)$ *and* $\bigwedge_{j=0}^{k} \varphi_j$ *are true in* $\mathcal{I}$. *Consequently, there is a run* $r = s_0 s_1 s_2 \ldots \in Runs(\mathcal{I})$ *in which we can find* $l \geq 0$ *such that* $\sigma_0 = s_l$ *satisfies*

$$\sigma_0 \sigma_1 \sigma_2 \ldots \models \phi_p \tag{4.1}$$

$$\sigma_0 \sigma_1 \sigma_2 \ldots \models \bigwedge_{j=0}^{k} \varphi_j \tag{4.2}$$

We have to show that $r \models \varphi_m$.

[4] For methodological purposes, we could assume that $p_{k+1} = \varepsilon$ and thus, the induction hypothesis corresponds precisely to the thesis.

We can split (4.2) into

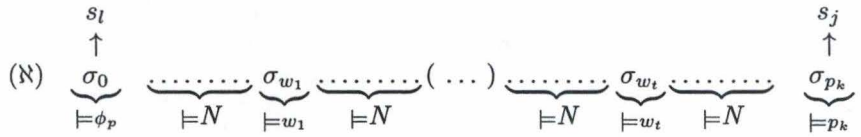
$$\sigma_0\sigma_1\sigma_2 \ldots \models \bigwedge_{j=0}^{k-1} \varphi_j \tag{4.3}$$

$$\sigma_0\sigma_1\sigma_2 \ldots \models \varphi_k \tag{4.4}$$

Using the induction hypothesis and (4.1), (4.3) can be rewritten

$$\sigma_0\sigma_1\sigma_2 \ldots \models AG(\phi_p \rightarrow \bigvee_{w \in \mathcal{L}_1 \cdot \ldots \cdot \mathcal{L}_k \cdot \{p_k\}} \phi_w) \tag{4.5}$$

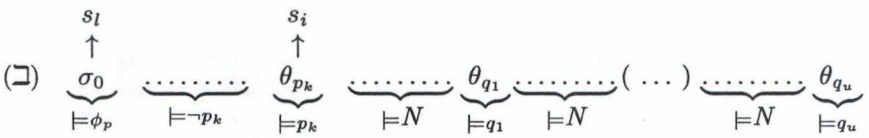
Since we know that (4.5) is true in $\mathcal{I}$, for every run rooted in σ_0, we can find a word $w \in \mathcal{L}_1 \cdot \ldots \cdot \mathcal{L}_k \cdot \{p_k\}$ satisfying ϕ_w. Let us assume that $w = w_1 \cdot \ldots \cdot w_t \cdot p_k$. Then, there are in the sequence of states $\sigma_1\sigma_2 \ldots$ some special states $\sigma_{w_1}, \ldots, \sigma_{w_t}, \sigma_{p_k}$ in which the corresponding event appears. The model of (4.5) would then be, graphically :



Now, let us develop (4.4), using its definition and (4.1).

$$\sigma_0\sigma_1\sigma_2 \ldots \models AG(\phi_p \rightarrow \bigvee_{w \in \mathcal{L}_{k+1}} \vartheta_w)$$

This statement holds, thus, for every run emanating from σ_0, we can find a word $q = p_k \cdot q_1 \cdot \ldots \cdot q_u \in \mathcal{L}_{k+1}$ such that $\phi_p \rightarrow \vartheta_q$ is true. Hence, in the sequence of states $\sigma_1\sigma_2 \ldots$ there are some special states $\theta_{p_k}, \theta_{q_1} \ldots, \theta_{q_u}$ in which the corresponding event appears. The model of (4.4) would then be, graphically:



If we show that the state σ_{p_k} in ℵ is the same state as θ_{p_k} in ℶ, we will be able to use ℵ and ℶ together to conclude.

So, we need to show that σ_{p_k} is the same state as θ_{p_k}. Or, using the indexes of the run $r = s_0 s_1 s_2 \ldots$, that $i = j$.

Let us assume that it is not the case, i.e. that $i \neq j$. There can be two possibilities, namely $i < j$ or $j < i$. We still have $i \geq l$ and $j \geq l$.

$l \leq i < j$.

We see in ℵ that s_i satisfies either N or a w_h, with $1 \leq h \leq t$.

$N \Rightarrow \neg p_k$, and thus, $s_i \not\models N$.

$\forall h : 1 \leq h \leq t : w_h \in$ pivot-prefix-lang$(\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m, p_k)$, hence, it is not possible to find a $w_h = p_k$, because, by the definition of a pivot (condition 2), $p_k \notin$ letters(pivot-prefix-lang$(\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m, p_k)$).

We should thus deduce that $j < i$.

$l \leq j < i$.

This situation is impossible, because, on the one hand $s_j \models p_k$ *and, on the other hand, the model* $\beth$ *shows clearly that,* $\forall h : l \leq h < j : s_h \not\models p_k$.

Therefore, we have to conclude that $i < j$.

We reach a contradiction, meaning that, necessarily, $s_i = s_j$.

Knowing that σ_{p_k} *and* θ_{p_k} *always denote the same state, we can put* $\aleph$ *and* $\beth$ *together :*

$$(\daleth) \quad \underbrace{\sigma_0}_{\models \phi_p} \underbrace{\ldots}_{\models N} \underbrace{\sigma_{w_1}}_{\models w_1} \underbrace{\ldots}_{\models N} (\ldots) \underbrace{\ldots}_{\models N} \underbrace{\sigma_{w_t}}_{\models w_t} \underbrace{\ldots}_{\models N} \underbrace{\sigma_{p_k} = \theta_{p_k}}_{\models p_k} \underbrace{\ldots}_{\models N} \underbrace{\theta_{q_1}}_{\models q_1} \underbrace{\ldots}_{\models N} (\ldots) \underbrace{\ldots}_{\models N} \underbrace{\theta_{q_u}}_{\models q_u}$$

and, using the definition of a pivot (condition 3), we are able to say that,

$$\underbrace{\underbrace{w_1 \cdot \ldots \cdot w_t}_{\in \textit{pivot-prefix-lang}(\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m, p_k)} \cdot p_k \cdot \underbrace{q_1 \cdot \ldots \cdot q_u}_{\in \textit{pivot-suffix-lang}(\mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m, p_k)}}_{\in \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m}$$

therefore, if we look at $\daleth$, *it comes that*

$$r \models \varphi_m.$$

□

The sufficient condition can be proved in a symmetric way. If we know that there is at least one run satisfying a trace of the prechart, we can reason backwards, using the model $\daleth$ *to go to the model* $\beth$. *Then, we can see immediately that*

$$\forall \mathcal{I} : \forall i : 0 \leq i \leq k+1 : \mathcal{I} \models \varphi \Rightarrow \mathcal{I} \models \varphi_i$$

□

Splitting up the formula (Method 2)

The first method we presented was intuitive but somehow too restrictive. Indeed, one of the characteristics of the pivots, that made them interesting, turns out to be a heavy disadvantage when splitting the formula. Just consider the following example.

Example 4.4 *Let the trace set of a chart be*

$$\mathcal{L} = \{ \; e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_1 \cdot e_2 \cdot e_3, \; e_1 \cdot e_2 \cdot e_4 \cdot e_3 \cdot e_1 \cdot e_2 \cdot e_3 \; \}$$

Clearly, only the first occurrence of e_2 *is a pivot in* $\mathcal{L}$ *that could be used to split up the corresponding formula. Following the first method, the formula describing this language could only be split into two formulae.*

Let us take a deeper look at what the models of the formula associated with $\mathcal{L}$ *would look like.*

$$\underbrace{\sigma_0}_{\models a} \underbrace{\ldots}_{\models N} \underbrace{\sigma_1}_{\models e_1} \underbrace{\ldots}_{\models N} \underbrace{\sigma_2}_{\models e_2} \underbrace{\ldots}_{\models (N \vee e_3 \vee e_4)} \underbrace{\sigma_3}_{\models e_1} \underbrace{\ldots}_{\models N} \underbrace{\sigma_4}_{\models e_2} \underbrace{\ldots}_{\models N} \underbrace{\sigma_5}_{\models e_3}$$

What keeps us from splitting the formula around σ_3 ?

Simply the fact that a state satisfying e_1 already appears before σ_3. Therefore, we cannot just say : "From a state satisfying a, ignore every state until you find a state satisfying e_1 and then restart the evaluation from this state", because we will not start again from the desired state, i.e. σ_3, but from σ_1.

Now, consider the following sequence of states :

$$\underbrace{\sigma_2}_{\models e_2}\ \underbrace{\cdots\cdots\cdots\cdots}_{\models (N \vee e_3 \vee e_4)}\ \underbrace{\sigma_3}_{\models e_1}\ \underbrace{\cdots\cdots}_{\models N}\ \underbrace{\sigma_4}_{\models e_2}\ \underbrace{\cdots\cdots}_{\models N}\ \underbrace{\sigma_5}_{\models e_3}$$

In this sequence, there is no problem to identify σ_3 anymore! So, what we will do is saying :"From a state satisfying a, first find a state satisfying e_2 and then a state in which e_1 appears, and start the evaluation from this state again.

The difference between this method and the first one is that we do not identify states with events anymore. Instead, *sequences of events* are used as identifiers. Now, we have to generalize this method.

First of all, we define inductively our extension of the pivots, that we call *extended pivots.* Then, we will propose a means to determine which sequence of events should be used to identify each extended pivots. Finally, we will be able to write a new split formula, using these extended pivots.

The following definition does not replace definition 4.2 but, instead, relies on it to define a less constraining concept, *extended pivots.*

Definition 4.8 (Extended pivots in a language $\mathcal{L}$) *In a finite language $\mathcal{L}$, we say that $(p_1, \ldots, p_k)$ are extended pivots if*

$$\mathcal{L}^0 = \mathcal{L}$$

$$p_1 \text{ is such that } \left(\begin{array}{l} p_1 \text{ is a pivot in } \mathcal{L}^0 \\ \wedge \text{ pivot-prefix-lang}(\mathcal{L}^0, p_1) \neq \{\varepsilon\} \\ \wedge \text{ pivot-suffix-lang}(\mathcal{L}^0, p_1) \neq \{\varepsilon\} \end{array} \right) (\alpha)$$

$$\wedge$$

$$\nexists\, p \,:\, p \text{ fulfills } \alpha \,\wedge\, p \prec p_1$$

$$\mathcal{L}^1 = \text{pivot-suffix-lang}(\mathcal{L}^0, p_1)$$

and, for every $i, 1 < i \leq k$

$$p_i \text{ is a pivot in } \mathcal{L}^{i-1} \text{ s.t. } \left(\begin{array}{c} \nexists p \,:\, p \text{ is a pivot in } \mathcal{L}^{i-1} \wedge p \prec p_i \\ \wedge \\ \text{pivot-suffix-lang}(\mathcal{L}^{i-1}, p_i) \neq \{\varepsilon\} \end{array} \right)$$

$$\mathcal{L}^i = \text{pivot-suffix-lang}(\mathcal{L}^{i-1}, p_i)$$

Here are some properties of these extended pivots, that are directly deduced from their definition.

Property 4.3 *Every p_i is a pivot in $\mathcal{L}^{i-1}$ but can also be a pivot in $\mathcal{L}^{i-2}, \mathcal{L}^{i-3}, \ldots, \mathcal{L}^s$, with $s \geq 0$. In particular, if p_i is an extended pivot and a pivot, then it is a pivot in $\mathcal{L}^0$, but also in $\mathcal{L}^1, \mathcal{L}^2, \ldots, \mathcal{L}^{i-1}$.*

Property 4.4 *Every p_i identifies uniquely its own* pivot-suffix *in $\mathcal{L}^{i-1}$, because, by definition, p_i is a pivot in $\mathcal{L}^{i-1}$.*

Property 4.5 *It is possible that $p_j = p_i$, for $j \neq i$, as example 4.5 will show.*

Properties 4.3 and 4.4 might seem to be *strengths* while the property 4.5 would more likely be a *weakness*. But this weakness is also a great asset, since it will allow us to avoid the limitation inherent to the use of pivots in the first method, namely that "an event may identify at most one state".

How can we optimally identify a p_i in $\mathcal{L}^0$? We could immediately use property 4.4. Indeed, we know that p_i is a pivot in $\mathcal{L}^{i-1}$, so it can be safely retrieved within $\mathcal{L}^{i-1}$. The problem is then reduced to identifying $\mathcal{L}^{i-1}$. But we know that it is uniquely identified by p_{i-1} in $\mathcal{L}^{i-2}$. And so on, until we get to $\mathcal{L}^0$.

So, we are sure that, for *every* p_i, there is actually a way to identify it. However, if we want to perform this task optimally, we have to use property 4.3.

Let us define recursively a new function $id(\mathcal{L}^i)$, which gives us the index of the *best* pivot from which we can identify $\mathcal{L}^i$. Here is a little example: if we have $e_1 \cdot e_2 \cdot e_1 \cdot e_3 \cdot e_1 \cdot e_4$, the shortest way to find the third e_1 is by using e_3. We jump from the beginning to e_3 and then, from e_3 to e_1.

We also define a function $length(\mathcal{L}^i)$ that gives us the number of *jumps* needed to retrieve p_i, which marks unambiguously the beginning of $\mathcal{L}^i$.

Definition 4.9 (*Id* **and** *length*)

$$\begin{aligned} length(\mathcal{L}^0) &= 0 \\ \\ id(\mathcal{L}^i) &= j \\ length(\mathcal{L}^i) &= length(\mathcal{L}^j) + 1 \end{aligned}$$

where

$$s \leq j < i \text{ with } \left(s = min\{k \mid p_i \text{ is a pivot in } \mathcal{L}^k\}\right) \wedge$$

$$\nexists k : (s \leq k < i) \wedge length(\mathcal{L}^k) < length(\mathcal{L}^j)$$

Now comes the definition of $seq(\mathcal{L}^j)$, a function returning the sequence of indexes of the pivots needed to identify $\mathcal{L}^j$.

Definition 4.10 (Sequence)

$$\begin{aligned} seq(j) &= j && \text{if } id(j) = 0 \\ seq(j) &= seq(\mathcal{L}^{id(\mathcal{L}^j)}) \cdot j && \text{otherwise} \end{aligned}$$

To illustrate these notions, we present a small example, based on the language of example 4.4

Example 4.5 *The definition 4.8 yields these successive results :*

$$\begin{aligned} \mathcal{L}^0 &= \{ \; e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_1 \cdot e_2 \cdot e_3, \\ &\qquad e_1 \cdot e_2 \cdot e_4 \cdot e_3 \cdot e_1 \cdot e_2 \cdot e_3 \} \\ p_1 &= e_2 \\ \mathcal{L}^1 &= \{ \; e_3 \cdot e_4 \cdot e_1 \cdot e_2 \cdot e_3, \\ &\qquad e_4 \cdot e_3 \cdot e_1 \cdot e_2 \cdot e_3 \} \\ \\ p_2 &= e_1 \\ \mathcal{L}^2 &= \{ \; e_2 \cdot e_3 \} \\ \\ p_3 &= e_2 \\ \mathcal{L}^3 &= \{ \; e_3 \} \end{aligned}$$

Why did we not define p_4 *? We can immediately see that* p_4 *would have had an empty pivot-suffix, which is forbidden, regarding to definition 4.8.*

Now, we have to determine the identifiers *and the* number of jumps *needed to retrieve each of the* p_i*'s.*

For p_1*, we have, as expected, that the only language in which it is a pivot is* $\mathcal{L}^0$*, so*

$$id(\mathcal{L}^1) = 0 \quad length(\mathcal{L}^1) = 1 \quad seq(\mathcal{L}^1) = 1$$

Regarding p_2*,* $\mathcal{L}^1$ *is the only language in which it is a pivot, yielding :*

$$id(\mathcal{L}^2) = 1 \quad length(\mathcal{L}^2) = length(\mathcal{L}^1) + 1 = 2 \quad seq(\mathcal{L}^2) = 1 \cdot 2$$

For p_3*, we have a choice, as we notice that it is a pivot in both* $\mathcal{L}^1$ *and* $\mathcal{L}^2$*. Therefore, we have to choose the identifier among* $\{p_1, p_2\}$ *that will* minimize *the number of jumps needed to reach* p_3*. It is* p_1 *since* $length(\mathcal{L}^1) < length(\mathcal{L}^2)$*.*

$$id(\mathcal{L}^3) = 1 \quad length(\mathcal{L}^3) = length(\mathcal{L}^1) + 1 = 2 \quad seq(\mathcal{L}^3) = 1 \cdot 3$$

The split formulae for an universal chart can now be redefined as

Definition 4.11 (Split universal formula)

$$\begin{aligned}
\varphi_0 &= AG\Big(\bigvee_{p \in \mathcal{L}^{trc}_{prech(m)}} \phi_p \rightarrow \bigvee_{w \in \mathcal{L}_{pp_1}} \rho^0_w\Big) \\
&\vdots \\
\varphi_i &= AG\Big(\bigvee_{p \in \mathcal{L}^{trc}_{prech(m)}} \phi_p \rightarrow \bigvee_{w \in \mathcal{L}_{pp_{i+1}}} \rho^i_w\Big) \\
&\vdots \\
\varphi_{k-1} &= AG\Big(\bigvee_{p \in \mathcal{L}^{trc}_{prech(m)}} \phi_p \rightarrow \bigvee_{w \in \mathcal{L}_{pp_k}} \rho^{k-1}_w\Big) \\
\varphi_k &= AG\Big(\bigvee_{p \in \mathcal{L}^{trc}_{prech(m)}} \phi_p \rightarrow \bigvee_{w \in \mathcal{L}^k} \rho^k_w\Big)
\end{aligned}$$

where, if we assume that $seq(\mathcal{L}^i) = i_1, \ldots, i_n, i$ *and* $w = p_i \cdot w'$*,*

$$\begin{aligned}
\mathcal{L}_{pp_1} &= pivot\text{-}prefix\text{-}lang(\mathcal{L}^0, p_1) \cdot \{p_1\} \\
\mathcal{L}_{pp_i} &= \{p_{i-1}\} \cdot pivot\text{-}prefix\text{-}lang(\mathcal{L}^{i-1}, p_i) \cdot \{p_i\}, \quad 1 < i \leq k
\end{aligned}$$

$$\begin{aligned}
\rho^0_w &= \phi_w \\
\rho^i_w &= (\neg p_{i_1})\, U\, (p_{i_1} \wedge X((\neg p_{i_2})\, U\, (p_{i_2} \wedge \ldots \wedge X((\neg p_i)\, U\, (p_i \wedge X(\phi_{w'})))\ldots))
\end{aligned}$$

One could easily show, using the same scheme as the one presented for the first method, that the following proposition still holds,

Theorem 4.2 (Equivalence of split and non-split formulae)

$$\forall\, \mathcal{I} \,:\, \mathcal{I} \models \bigwedge_{j=0}^{k} \varphi_j \quad \Longleftrightarrow \quad \mathcal{I} \models \varphi$$

The proof would be perfectly identical, except the part where we show that, thanks to the definition of *extended pivot*, see definition 4.8, a state can be safely retrieved.

In most cases, this method will yield redundant formulae. Before model-checking them, one should first get rid of all the formulae φ_i, such that $\varphi_j \Rightarrow \varphi_i$, for a $j > i$.

This verification could be done efficiently using a model-checker, and looking for the validity of $\varphi_j \Rightarrow \varphi_i$. By now, we have the intuition that checking this for $j = i + 1$ should be enough but this claim is still to be proved.

4.2 Computing pivots in finite languages

The aim of our work is to provide methods and algorithmic solutions that could be integrated into a tool for automatic verification of LSCs. Following this idea, we have to answer the following question: *How can we algorithmically find pivots in finite languages?* .

Remember that pivots have been defined for *finite* languages. They are regular languages, that can be recognized by *acyclic* deterministic finite automata (ADFA). An ADFA is a DFA, so, every well-known result holding for DFA also applies to ADFA.

In the remainder of this section, we use minimal automata. We recall their definition.

Definition 4.12 (Minimal DFA) *A DFA A is called* minimal *if there is no DFA A' such that there is strictly less states in A' than in A and $\mathcal{L}(A) = \mathcal{L}(A')$.*

There are well-known algorithms to transform any DFA into a minimal DFA, see [Wilhelm 94, Aho 00].

4.2.1 Characterization of pivots in finite languages

We translate the definition of pivots (definition 4.2), given in terms of languages, into concepts suited to automata. First, we define remarkable states, that we call *pivot-states* and, afterwards, we show that these remarkable states correspond to pivots in the language recognized by their automaton.

Definition 4.13 (Pivot-state) *In an ADFA $< \Sigma, S, s_0, \rho, f >$, we say that $s \in S$ is a* pivot-state *for a letter $p \in \Sigma$ if s satisfies the following conditions*

$$\forall s' : (s', x, s) \in \rho \Rightarrow x = p \qquad (1)$$

$$\wedge$$

$$\left[\begin{array}{l} \forall \text{ accepting path } (s_0 \ldots s_i \ldots s_n) \text{ in } A, \\ \\ \exists i : 1 \leq i \leq n : \left(\begin{array}{c} s_i = s \\ \wedge \\ \forall j : 1 \leq j < i : (s_{j-1}, p, s_j) \notin \rho \end{array} \right) \end{array} \right] \qquad (2)$$

It is noteworthy that condition (2) implies that $s \neq s_0$. Intuitively, one sees that this definition is close to the definition of a pivot since

1. it obliges every accepting path to use this state, as a pivot letter obliged every word to use it.

2. it forbids any occurrence of the letter p before its incoming transition, as a pivot letter could not occur in its own pivot-prefix, either.

The same process can be applied to definition 4.8, to provide us with a definition for extended-pivots, but in automata-related terms.

Definition 4.14 (Extended Pivot-states) *In an ADFA $< \Sigma, S, s_0, \rho, f >$, we say that $(s^0, \ldots, s^k)$ are* extended-pivot-states *for $(p_0, \ldots, p_k)$, if they fulfill the*

following inductive definition

$$\left[\begin{array}{l} \textit{for every accepting path } (s_0 \ldots s_i \ldots s_n) \textit{ in } A, \\ \exists i_0, \ldots, i_k \ : \ \bigwedge_{j=0}^{k} s_{i_j} = s^j \\ \qquad\qquad \wedge\ i_0 < \ldots < i_k \end{array}\right] \tag{1}$$

$$\wedge$$

$$\left[\begin{array}{l} s^0 \textit{ is a pivot-state in } A \textit{ for } p_0 \\ \forall i : \ 0 < i \leq k : \ s^i \textit{ is a pivot-state in } < \Sigma, S, s^{i-1}, \rho, s^i > \textit{ for } p_i. \end{array}\right] \tag{2}$$

The algorithmic solution presented later is based on this characterization.

The following theorem links the concept of *pivot-state in an automaton* to the concept of *pivot in a language.*

Theorem 4.3 (Pivot in a minimal ADFA)

Let $A = < \Sigma, S, s_0, \rho, f >$ *be a* minimal acyclic *DFA.*

p is a pivot in $\mathcal{L}(A)$

$$\Updownarrow$$

$\exists!\ s \in S$ *such that*

s is a pivot-state for p in A

Theorem 4.4 (Extended-Pivot in a minimal ADFA)

Let $A = < \Sigma, S, s_0, \rho, f >$ *be a* minimal acyclic *DFA.*

$(p_0, \ldots, p_k)$ *are extended-pivots in* $\mathcal{L}(A)$

$$\Updownarrow$$

$\exists!\ (s^0, \ldots, s^k) \in S^*$ *such that*

$(s^0, \ldots, s^k)$ *are extended pivot-states for* $(p_0, \ldots, p_k)$ *in A*

For the sake of brevity, we do not prove this last theorem. We will only prove theorem 4.3.

Proof 4.6 (Pivots in a minimal ADFA) *We will first prove the necessary condition.*

To do so, we assume that p is a pivot in $\mathcal{L}(A)$ *and we have to show that there is a unique state* $s \in S$ *satisfying conditions (1) and (2) of definition 4.13. For the remainder of this proof, (1) and (2) refer respectively to the first and second condition of definition 4.13.*

Note that the uniqueness of s is deduced from the determinism of A and the characterization of s. Indeed, it is impossible to find two states in a DFA satisfying these two conditions, for the same pivot p. If A is deterministic then there cannot be two different accepting paths for one word. Hence, if there were two states satisfying (2), for the same given p, one should precede the other, which, clearly, when looking at (1), violates (2).

It will thus be enough to prove that s exists.

First of all, we show that there is a state s fulfilling (1). For the sake of contradiction, we assume that for every state in S, there is at least one incoming transition not labelled by *p. Thus, we are able to follow an accepting path from* s_0 *to f that does not use any transition labelled by p. Hence, this path defines a word belonging to* $\mathcal{L}(A)$ *that contains no occurrence of p. Since p is a pivot, this statement violates the second condition of definition 4.2.*

We reach a contradiction, so we have to conclude that there is at least one state $s \in S$ *satisfying (1).*

Now, we prove that there is a s among the states fulfilling the condition (1) that satisfies (2), too.

Let s and s' be two states satisfying (1). We assume that there are two different accepting paths in A, fulfilling the requirements of (2), one using s and the other one using s'.

These two paths define two different words in $\mathcal{L}(A)$ *:*

$$w_1 = \alpha \cdot p \cdot \lambda \quad s.t. \quad \alpha = \textit{pivot-prefix}(w_1, p)$$
$$w_2 = \beta \cdot p \cdot \mu \quad s.t. \quad \beta = \textit{pivot-prefix}(w_2, p)$$

with $\alpha \neq \beta$, $w_1 \neq w_2$, $\alpha \cdot p$ *leading to s and* $\beta \cdot p$ *leading to s'.*

The "$\alpha =$ *pivot-prefix*(w_1, p)*" statement (sym. for* β*) is deduced from the fact that the two paths that we chose respect the requirement (2) and that s, as well as s', fulfills condition (1).*

Since w_1 *and* w_2 *are in* $\mathcal{L}(A)$*, and p is a pivot in* $\mathcal{L}(A)$*, we have* $\alpha \cdot p \cdot \mu \in \mathcal{L}(A)$ *and* $\beta \cdot p \cdot \lambda \in \mathcal{L}(A)$*.*

A is deterministic, consequently, $\alpha \cdot p$ *leads to s and* $\beta \cdot p$ *leads to s'.*

Therefore, from both s and s', A is able to recognize λ *and* μ*. Hence, we have to conclude that A is not minimal since we can build another automaton* $A' = < \Sigma, S \setminus \{s'\}, s_0, \rho', f >$ *with*

$$(\sigma, a, \sigma') \in \rho' \iff (\sigma, a, \sigma'') \in \rho \wedge \begin{pmatrix} (\sigma'' \neq s') \Rightarrow \sigma' = \sigma'' \\ \wedge \\ (\sigma'' = s') \Rightarrow \sigma' = s \end{pmatrix}$$

such that A' *has less states than A but, still,* $\mathcal{L}(A') = \mathcal{L}(A)$*.*

Since we reach a contradiction with the hypothesis that A is minimal, we should conclude that the assumption that there was an accepting path not using s is falsified, so, every accepting path uses *s.*

This finishes the demonstration of the necessary condition.

□

To demonstrate the sufficient condition, we have to show that p fulfills every condition defining a pivot.

For the first condition, it is straightforward. We know that A is deterministic. It means that there is no ε*-transition in A, so* $p \neq \varepsilon$*.*

The second condition states that every word of $\mathcal{L}(A)$ contains at least one occurrence of p. Using (2), it is trivially verified.

The third condition is as easy to prove as the first two.
We take two accepting paths in A,

$$\begin{array}{ll} s_0\, s_1 \ldots s_{i-1}\, (s = s_i)\, s_{i+1} \ldots s_m\, f & \textit{accepting the word } \alpha \cdot p \cdot \lambda \\ s_0\, s'_1 \ldots s'_{k-1}\, (s = s'_k)\, s'_{k+1} \ldots s'_n\, f & \textit{accepting the word } \beta \cdot p \cdot \mu \end{array}$$

Of course, these two paths are also accepting,

$$\begin{array}{ll} s_0\, s_1 \ldots s_{i-1}\, (s = s_i)\, s'_{k+1} \ldots s'_n\, f & \textit{accepting the word } \alpha \cdot p \cdot \mu \\ s_0\, s'_1 \ldots s'_{k-1}\, (s = s'_k)\, s_{i+1} \ldots s_m\, f & \textit{accepting the word } \beta \cdot p \cdot \lambda \end{array}$$

and this last deduction corresponds to the statement of the third condition.

□

4.2.2 An algorithm to find pivots in finite languages

Algorithm 3 is based on theorem 4.3. It simply "scans" the automaton to find all the states satisfying definition 4.13. Here is an informal description of this algorithm.

Given a minimal ADFA A, we first compute the shortest accepting path in A. We know that every accepting path uses all the pivot-states. Consequently, every state beside s_0 in this path is a potential pivot-state.

From this list of candidates, we remove the states that do not satisfy condition (1) in definition 4.13. We simply check in the transition relation that all the transitions ending in the same state bear the same label.

Then, for every candidate state remaining in the list, we see if there is still an accepting path in A when this state is removed. If it is the case, then this state is surely not a pivot-state, since it violates condition (2) of definition 4.13. We must thus remove this state from the list.

Finally, we check that every candidate pivot does not appear in its own pivot-prefix.

In order to do so, we define a more general problem. We consider that the automaton has been *sliced into zones*, such that zone i contains all the states and the transitions[5] between the candidate pivot-state s_{i-1} and s_i. By *candidate pivot-state*, we mean a state in which all the incoming transitions bear the same label and that appears on every accepting path. The max-prefix subproblem consists of labelling every candidate pivot-state with a set of couples (e, j) where $e \in \Sigma$ and $0 < j \leq i$ or $j = \mathsf{nil}$. A label (e, j) means that letter e appears in zone j and does not appear in any zone between zone j and the zone of the current candidate pivot-state. A label (e, nil) means that e did not appear in any zone preceding the current zone.

Example 4.6 (Max-Prefix) *Figure 4.5 shows an automaton divided into zones. Note that a zone i does not include the transitions ending in the candidate pivot-state sc_i.*

Max-prefix should label the candidates pivot-states with the following couples:

sc_1: $(a, \mathit{nil}), (b, \mathit{nil}), (c, 1), (d, 1)$

sc_2: $(a, 2), (b, \mathit{nil}), (c, 1), (d, 1)$

sc_3: $(a, 3), (b, 3), (c, 3), (d, 1)$

[5] Except the last one

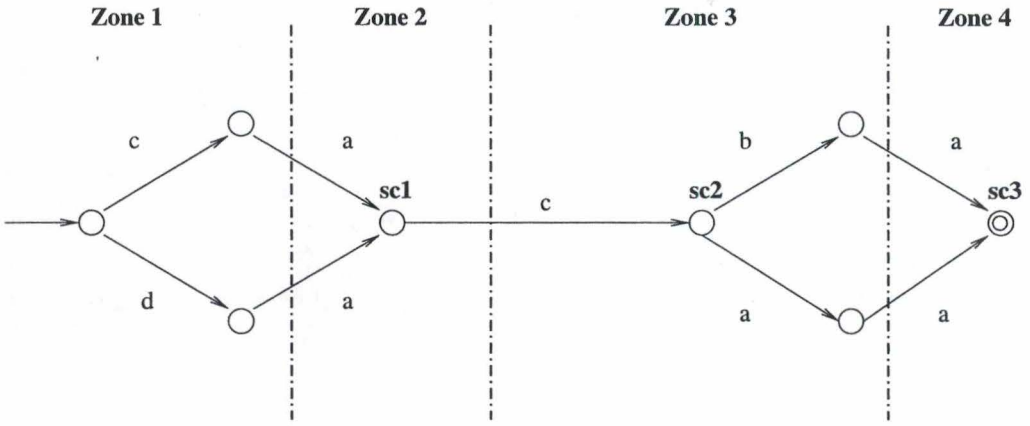


Figure 4.5: An automaton divided into four zones

Of course, we can use these labels to determine if sc_i is a pivot-state. If the letter for which sc_i could be a pivot-state is labelled by nil, we know that this letter did not appear on any transition before the transitions ending in sc_i. Therefore, sc_i is a pivot-state. For example, sc_1 is a pivot-state while neither sc_2 nor sc_3 are.

Max-prefix provides us with enough information to also detect extended pivot-states. In fact, if the letter for which sc_i could be an extended pivot-state is labelled by a zone number that is stricly smaller than i, then, sc_i is an extended pivot-state. For example, sc_1 and sc_2 are extended pivot-states, while sc_3 is not.

Our algorithm is general enough to be used for both method 1 and method 2. The solution to the subproblem *max-prefix* provides us with all the information needed to compute the *id* and *length* functions, if we want to use the second method.

Indeed, for every extended pivot-state, *lastappear*, the labelling of zones provided by max-prefix, gives us the value of s (the most including language in which this extended pivot is a pivot, see definition 4.9)

We would like to bring the reader's attention to the fact that this algorithm takes as input an ADFA with only *one* accepting state. This restriction introduces no loss of generality.

If the automaton of which we were trying to compute pivots had more than one final state, the first list of candidates for pivot states would just be the largest common prefix of every accepting path. In other words, the (largest) common beginning of the shortest paths from the initial state to every final state.

4.2.3 Complexity issues

Time complexity of algorithm 3 is $O(|S|^2 + |S|\ |\rho|)$. This comes from the fact that breadth-first search is in $time(O(|S| + |\rho|))$, see [Cormen 96], and that the BFS-ADFA subproblem is used at most $|S| + 1$ times.

Every other subproblem used has a lower time complexity. In particular, the time complexity of max-prefix is $O(|S| + |\rho|)$.

Space complexity is $O(|S|^2 log\,|S|)$ because we use work space to store *lastappear*, which can be seen as a matrix of size $|S| \times |S|$, in which each cell stores a number bounded by $|S|$ (the maximum number of pivots).

Algorithm 3 Find pivots in a finite language - Find-Pivots(A)

Input : a minimal ADFA $A =< \Sigma, S, s_0, \rho, f >$.

Output : *listps*, the longest possible list of pivot-states for $\mathcal{L}(A)$, if we use method 1, or extended pivot-states, if we use method 2. They are ordered according to the precedence order of pivots.

if BFS-DFA(A) outputs NO **then**
  return ()

else {there is an accepting path in A}

  listps $\leftarrow$ BFS-DFA(A)
  remove s_0 from *listps*

  for all $s \in$ *listps* such that $|\{e \mid \exists s' \in S : (s', e, s) \in \rho\}| > 1$ **do**
    remove s from *listps*
  end for

  for all $s \in$ *listps* **do**
    $A' =< \Sigma, S \setminus \{s\}, s_0, \rho', f >$
    {where ρ' is ρ modified to stay consistent with the removal of s from S}

    if BFS-DFA(A') does not output NO **then** {there is an accepting path not using s}
      remove s from *listps*
    end if
  end for

  lastappear $\leftarrow$ max-prefix(A, *listps*)
  Let *listps* $= [s_1, \ldots, s_n]$ and $[p_1, \ldots, p_n]$ be the labels of the transitions ending in these states.
  for all $s_i \in$ *listps* **do**
    if

$$\begin{array}{rll} \textit{lastappear}\,[i][p_i] & \leq i & \text{(Method 1)} \\ & = i & \text{(Method 2)} \end{array}$$

    then
      remove s_i from *listps*
    end if
  end for

  {Termination}
  return *listps*
end if

Algorithm 4 Breadth-First Search in a DFA - BFS-DFA(A)

Input : a DFA $A =< \Sigma, S, s_0, \rho, F >$.

Output : The sequence of states on the shortest path from the initial state s_0 to a final state ($\in F$) of A, or NO if there is no *accepting path* in A (and thus, $\mathcal{L}(A) = \emptyset$).

$\{\pi : S \rightarrow S + \{\text{nil}\}$, the predecessor of each state on the shortest path from $s_0\}$
$\{R : S^*$, the queue of states still to be developed$\}$

{Initialization}
for all $s \in S$ **do**
  $\pi[s] \leftarrow$ nil
end for
$R \leftarrow (s_0)$

{Iteration}
while $(R \neq ()) \wedge (\forall f \in F, \pi[f] = \text{nil})$ **do**
  {Remove the first state in R and put it in s}
  $s \leftarrow head(R)$
  $R \leftarrow tail(R)$

  for all $s' \in S$ such that $\exists a \in \Sigma : (s, a, s') \in \rho$ **do** {s' is a successor of s}
    if $\pi[s'] =$ nil **then** {s' has not been developed yet}
      $\pi[s'] \leftarrow s$
      $R \leftarrow R \cdot (s')$
    end if
  end for

end while

{Termination}
if $\forall f \in F, \pi[f] =$ nil **then** {there is no accepting path}
  return NO
else {the shortest path has been found}
  choose $f \in F$ such that $\pi(f) \neq$ nil
  $l \leftarrow (f)$
  $s \leftarrow f$
  while $s \neq s_0$ **do**
    $s \leftarrow \pi[s]$
    $l \leftarrow s \cdot l$
  end while
  return l
end if

Algorithm 5 Maximum prefix in which a letter appears - max-prefix($A, listps$)

Input : a ADFA $A =< \Sigma, S, s_0, \rho, f >$.
$listps = [sc_1, \ldots, sc_n]$ a list of states such that, $\forall i : 1 \leq i \leq n$,
every accepting path in A uses sc_i
and all the transitions ending in sc_i bear the same label

Output : a function giving, for every letter e and state sc_i, the biggest j, such that $1 \leq j \leq i$ and the letter e labels a transition on a path between sc_{j-1} and sc_j, ignoring any transition ending in sc_i.

$\{\diamond \notin \Sigma$ and $\Sigma' = \Sigma + \{\diamond\}\}$
$\{lastappear \in \mathbb{N} \rightarrow (\Sigma' \rightarrow \mathbb{N} + \{\mathsf{nil}\})\}$

{Initialization}
$s' \leftarrow s_0$
$f' \leftarrow sc_1$
$i \leftarrow 1$

for all $j \in \mathbb{N}, e \in \Sigma'$ **do**
  $lastappear\,[j][e] \leftarrow$ nil
end for

{Iteration}
while $i < n$ **do**
  Let $\rho' = \rho$ in which the transitions ending in s' are labeled by $\diamond$
  $lastappear\,[i] \leftarrow$ DFS-letters$(< \Sigma', S, s', \rho', f' >, lastappear\,[i-1], i)$
  $lastappear\,[i+1][p] \leftarrow i$
  {where p is the label of the transitions ending in s'}
  $i \leftarrow i+1$
  $s' \leftarrow sc_{i-1}$
  $f' \leftarrow sc_i$
end while

{Termination}
return *lastappear*

Algorithm 6 Compute $letters(\mathcal{L}(A))$ - DFS-letters$(A, letters, k)$

Input : a DFA $A =< \Sigma, S, s_0, \rho, f >$.
$letters : \Sigma \to \mathbb{N}$.
$k \in \mathbb{N}$

Output :

$$\forall e \in \Sigma : \quad (e \in letters(\mathcal{L}(A)) \Rightarrow letters[e] = k)$$
$$(e \notin letters(\mathcal{L}(A)) \Rightarrow letters[e] \text{ is unchanged})$$

{$visited : S \to \mathbb{B}ool$}
{$R : S^*$, the list of states still to be developed}

{Initialization}
$R \leftarrow (s_0)$
for all $s \in S$ **do**
  $visited[s] \leftarrow$ false
end for

{Iteration}
while $R \neq ()$ **do**
  {Remove the first state in R and put it in s}
  $s \leftarrow head(R)$
  $R \leftarrow tail(R)$
  $visited[s] \leftarrow$ true

  for all $e \in \Sigma, s' \in S$ **do**
    if $(s, e, s') \in \rho$ **then** {s' is a successor of s on letter e}
      $letters[e] \leftarrow k$
      if $\neg(visited[s'])$ **then** {s' has not been developed yet}
        $R \leftarrow (s') \cdot R$
      end if
    end if
  end for

end while

{Termination}
return $letters$

4.3 Computing pivots in trace sets

4.3.1 Automaton-based technique

On the one hand, we have shown that the size of an automaton recognizing the trace set of a chart had a number of states on the order of an exponential of the size of the chart. On the other hand, the algorithmic solution to the computation of pivots used minimal *deterministic* automata while our translation algorithm produced *non-deterministic* automata. Therefore, we would have to determinize our automaton. This operation may cause an exponential blow-up of its number of states. The conclusion is thus that computing pivots in trace sets has a *double exponential* complexity.

Theorem 4.5 (Complexity of finding pivots in trace sets) *In order to find the pivots in a trace set, we should*

1. *build* $A_{cuts(m)}$ *in time*$(O(2^{|dom(m)|}))$.
2. *determinize* $A_{cuts(m)}$ *in time*$(O(2^{2^{|dom(m)|}}))$.
3. *minimize* $A_{cuts(m)}$ *in time*$(O(2^{2^{|dom(m)|}}))$, *since* $A_{cuts(m)}$ *is acyclic, and thus, the minimization can be done efficiently, in time linear in the size of the automaton.*
4. *find the pivots in it with the same time complexity,* $O(2^{2^{|dom(m)|}})$.

Now, we replace the general characterization of pivots in ADFA (theorem 4.3) by the following, stronger, condition. It is easier to program an algorithm verifying this condition. This algorithm will be a little bit more efficient than the general algorithm 3 presented above, as its time complexity will be linear in the size of the automaton instead of quadratic. However, the whole problem of finding pivots in trace sets will have a double-exponential time complexity.

Theorem 4.6 (Pivots in $\mathcal{L}_m^{trc}$) *Let* $(c_1^s, \ldots, c_m^s)$ *be the largest common prefix of the shortest paths from* $\{0_1, \ldots, 0_n\}$ *to every final cut. We suppose that* $A =< \Sigma, S, s_0, \rho, F >$ *is the minimal acyclic deterministic automaton recognizing* $\mathcal{L}_m^{trc}$.

$$
\begin{array}{c}
p \text{ is a pivot in } \mathcal{L}_m^{trc} \\
\Updownarrow \\
\exists\, c_1, c_2 \in c_1^s, \ldots, c_m^s \\
\left(\begin{array}{ll}
 & (c_1, p, c_2) \in \rho \\
\wedge & \nexists p', c' \ : \ (c', p', c_2) \in \rho \wedge c' \neq c_1 \\
\wedge & \nexists p', c' \ : \ (c_1, p', c') \in \rho \wedge c' \neq c_2 \\
\wedge & \nexists l \in c_1 \ : \ p = ev(l)
\end{array} \right)
\end{array}
$$

Intuitively, this theorem means that p is a pivot in a trace set if there are two cuts c_1, c_2 always appearing on every accepting path, such that c_1 is the only predecessor of c_2 and c_2 is the only successor of c_1. We add two conditions:

1. p is the event labelling the transitions between those cuts
2. p did not appear before, on this accepting path.

Proof 4.7 (Theorem 4.6) *Proving the sufficient condition is straightforward, for* c_2 *is obviously a pivot-state, and we can thus apply theorem 4.3.*

The necessary condition is shown using theorem 3.2.

We prove the necessary condition as follows. We suppose that p is a pivot in $\mathcal{L}_m^{trc}$ *and, applying theorem 4.3, we assume that* c_2 *is a pivot-state.*

The first and last part of the necessary condition are straightforwardly derived from the fact that c_2 *is a pivot state.*

Of course, we have that $\forall c \in cuts(m) : (c, p', c_2) \in \rho \Rightarrow p' = p$.

We demonstrate now that c_2 *has at most one predecessor. The proof is done by contradiction, making the assumption that there are two distinct cuts* c, c' *such that* $(c, p, c_2) \in \rho$ *and* (c', p, c_2). *Then, by theorem 3.2, it comes that there are two locations* l, l' *such that p is the event associated with both of them and* $l' \in c$ *and* $l \in c'$. *If* $l' \in c$, *then location* l' *has already been visited before reaching c. Therefore, we should conclude that p cannot be a pivot (since it already appeared on the accepting path), which violates our primary hypothesis. Hence, c and* c' *may not be distinct.*

We should now show that c_1 *has one and only one successor,* c_2. *If we assume that it is not the case, that is, there is an a-transition to another cut c from* c_1, *with* $c_2 \neq c$ *and* $a \neq p$, *we raise a contradiction with the fact that* c_2 *is a pivot state. Indeed, on the one hand, we have that* $c_2 = c_1 \cup \{l_p\}$, *where* l_p *and* l_a *are the locations to which the events p and a are respectively attached. We also have that* $c = c_1 \cup \{l_a\}$. *On the other hand, we have that both c and* c_2 *appear on an accepting path as, clearly, there is no* not *accepting paths in a trace set's automaton. We can deduce that c belongs to an accepting path on which* c_2 *does not appear, because the number of locations in* c_2 *is the same as the number of locations in c. Thus, clearly,* c_2 *could not appear* after *c on an accepting path. Therefore, we showed that* c_2 *cannot be a pivot state, which is impossible.*

□

4.3.2 Non Automaton-based technique

From the facts presented above, we have to conclude that the complexity of computing pivots in a trace set is related to the requirement of using a minimal ADFA as a basis for the computation.

Thus, it could be interesting to relax this constraint. We could try to use the causal order $<_m$ to identify the pivots. Hence, we must find a way to link the concept of pivot or pivot-state to the concept of location in a causal order.

We would rely on the intuition that a pivot p corresponds to a location l such that

1. $ev(l) = p$
2. $\forall l' \in dom(m), (l <_m l') \vee (l' <_m l)$

This location fulfills the constraint that every other location should appear either *always before* or *always after* it.

If we take a more precise look at this requirement, we see that is not sufficient, because it does not express that a pivot letter may never appear in its own prefix. Hence, we have to ensure that there is no location l' such that $(l' <_m l) \wedge (ev(l) = ev(l') = p)$.

These three conditions could be checked in time linear in the size of $<_m$.

A pivot must appear in every word of its language, too. This condition looks a little bit more complicated to check, as we have to verify that no final cut (i.e. a final state in the automaton) could be built only with locations preceding l. However, this condition could be checked using $\mathrm{A}_{cuts(m)}$, the ANFA recognizing the trace set of the chart.

Consequently, it could be possible to find pivots in a trace set in time exponential in the size, if we are obliged to use ANFA in order to check for the last condition.

We ran out of time to prove this relationship between locations and pivots. This result is thus intuitively obvious but is not sustained by any formal proof.

4.4 Translating charts into LTL and CTL

LSCs can be translated into formulae that belong to sublogics of CTL*. In particular, research work highlighted that they could be transformed into LTL and CTL. An article to come, [Kugler 01], will formally prove this equivalence, for a subset of our LSC language.

These formulae allow us to use a wide range of model-checkers to perform the verification task. As a matter of fact, there are few CTL* model-checkers but there are a lot of software devoted to the verification of CTL or LTL formulae. Thus, we will be able to combine our verification software with the most efficient tool, among a lot of different available model-checkers.

As usual, we will not pay any particular attention to *no-stories*, as they are nothing but negated existential chart.

4.4.1 LTL formulae

LTL: Definition

LTL, or Linear Temporal Logic, is a subset of CTL*, which provides operators for describing events along a single computation path.

It consists of formulae having the form Af, where f is an LTL path formula, as defined below.

Definition 4.15 (LTL path formula) Linear Time Logic path formulae *are CTL* path formulae restricted to the following form:*

1. *if $f \in Prop$ then f is a LTL path formula*
2. *if f, g are LTL path formulae, then*

   $\neg f$

   $f \vee g$

   $f \wedge g$

   Xf

   Ff

   Gf

   gUf

   are also LTL path formulae.

Translating a universal chart into an LTL formula

The CTL* formula for a chart, as definition 3.4 states, is of the form

$$\varphi_m = AG(\bigvee_{w \in \mathcal{L}^{trc}_{prech(m)}} \phi_w \rightarrow \bigvee_{w \in \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m}} \phi_w)$$

with

$$\phi_{e_1 \cdot e_2 \cdots e_n} = (e_1 \wedge X(N U (e_2 \wedge X(N U (e_3 \wedge X(\ldots X(N U e_{n-1} \wedge X(N U e_n) \ldots)$$

One can immediately see that ϕ_w is an LTL path formula and, consequently,

$$\alpha_m = G(\bigvee_{w \in \mathcal{L}^{trc}_{prech(m)}} \phi_w \quad \rightarrow \quad \bigvee_{w \in \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m}} \phi_w)$$

is also an LTL path formula.

Therefore, $A\ \alpha_m$ is an LTL formula.

Translating an existential chart into an LTL formula

The general CTL* formula describing an existential chart m is (definition 3.5)

$$\varphi_m \ = \ EF(\bigvee_{w \in \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m}} \phi_w)$$

Of course, the following formula is an LTL path formula:

$$\beta_m = F(\bigvee_{w \in \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_{m}} \phi_w)$$

Since the E temporal operator is the dual of A, it comes that $\neg E \neg f \iff Af$.

We still have that $\neg\beta_m$ is an LTL path formula, and consequently, $A\neg\beta_m$ is an LTL formula.

A system $\mathcal{I}$ would thus satisfy the existential chart m if, and only if,

$$\mathcal{I} \not\models A\neg\beta_m$$

4.4.2 CTL formulae

CTL is a subset of CTL* formulae in which the temporal operators quantify over the paths that emanate from a given state. In other words, in CTL, every operator X, U, F, G should be path quantified by either A or E.

Translating a universal chart into a CTL formula

The general CTL* formula for an universal is not a CTL formula. Transforming it into a valid CTL formula turns out to be quite tricky.

In first approximation, one should be aware that only distributing an A quantifier over every operator in the formula does not work, as this will oblige the same trace to hold on every computation path.

A good approach would be to build a formula f_m such that f_m describes the violation of the universal chart m. Then, we would check for $\mathcal{I} \not\models f_m$, as every system that does not violates the chart does satisfy it.

What is a violation of a universal chart? It is a trace that satisfies the prechart but does not always satisfy the main chart.

It is quite difficult to express "*does not always satisfy the main chart*".

In order to do so, we have to enrich a little bit the formula, by making it possible to reflect the structure of the chart. Hence, we need to use *Tree Automata*, instead of *Acyclic Automata.*

Definition 4.16 (Tree Automaton) *An acyclic automaton* $< \Sigma, S, s_o, \rho, F >$ *is called a* Tree Automaton *if it satisfies the following conditions:*

1. $\forall s \in S \setminus \{s_0\} : |\{s' \in S | \exists e \in \Sigma \ : \ (s', e, s) \in \rho\}| = 1$
2. $\forall s : (\nexists s' \in S, e \in \Sigma \ : \ (s, e, s') \in \rho) \iff s \in F$

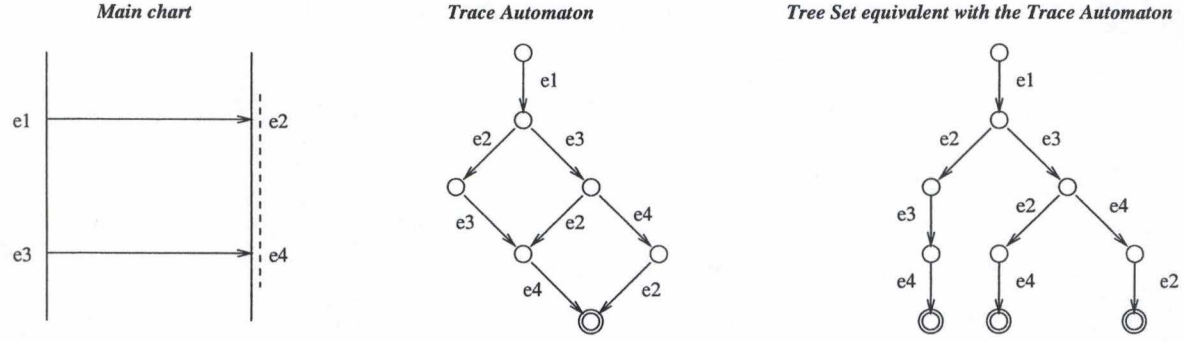


Figure 4.6: A main chart m, $A_{cuts(m)}$ and $Tree(A_{cuts(m)})$

The first condition means that every state has at most one predecessor, while the second condition states that only final states have no successors.

Every acyclic automaton can be turned into an equivalent tree automaton, i.e. a tree automaton recognizing the same language.

We give a function that *builds* a CTL state formula for a given tree automaton. The tree automaton is of course equivalent with the trace set automaton of the considered chart.

This formula expresses that, from the state considered, every computation path satisfies one of the traces belonging to the tree automaton.

Definition 4.17 (γ)

$$\gamma(< \Sigma, S, s_0, \rho, F >) = AX(NAU \bigvee_{i=1}^{n} (e_i \wedge \gamma(A_i)))$$

where

$\{(e_1, s_1), \ldots, (e_n, s_n)\} = \{(e, s) |\ (s_0, e, s) \in \rho\}$

$A_i =< \Sigma, S, s_i, F >$

Now, we want to express that a given trace of the prechart is satisfied and leads to a state s from which, afterwards, the main chart is violated.

Definition 4.18 (δ_i^m)

$\delta^m_{e_1 \cdot e_2 \cdot \ldots \cdot e_n} = e_1 \wedge EX(NEU(e_2 \wedge (EX \ldots \wedge EX(NEU(e_n \wedge \neg\gamma(Tree(A_{cuts(m)})) \ldots)$

The CTL formula for a given chart m is thus

$$\varphi_m = \neg \bigvee_{w \in \mathcal{L}^{trc}_{prech(m)}} \delta^m_w$$

We show how the translation works on a small example.

Example 4.7 (CTL formula for a universal chart) *Given the main chart of figure 4.6, the γ function would yield the following CTL formula:*

$$\begin{array}{lll} e_1 \wedge AX(N\ AU & (e_2 \wedge AX(N\ AU & (e_3 \wedge AX(N\ AUe_4)))) \\ & \vee & \\ & (e_3 \wedge AX(N\ AU & (e_2 \wedge AX(N\ AUe_4)) \\ & & \vee \\ & & (e_4 \wedge AX(N\ AUe_2)) \end{array}$$

$)$

In order to express the semantics of an activated chart, this formula should of course be integrated into a longer formula, checking the validity of the run, with respect to the prechart.

Translating an existential chart into a CTL formula

We simply have to check that there exists a computation path on which one of the traces hold.

In the formula

$$\phi_{e_1 \cdot e_2 \cdots e_n} = (e_1 \wedge X(N\,U\,(e_2 \wedge X(N\,U\,(e_3 \wedge X(\ldots X(N\,U\,e_{n-1} \wedge X(N\,U\,e_n)\ldots)$$

we simply quantify every operator with an E. The formula becomes

$$\phi^{CTL}_{e_1 \cdot e_2 \cdots e_n} = (e_1 \wedge EX(N\,EU\,(e_2 \wedge EX(N\,EU\,(e_3 \wedge EX(\ldots EX(N\,EU\,e_{n-1} \wedge EX(N\,EU\,e_n)\ldots)$$

and the general formula remains unchanged, except that we replace ϕ_w by its CTL equivalent ϕ^{CTL}_w:

$$\varphi_m = EF(\bigvee_{w \in \mathcal{L}^{trc}_{prech(m)} \cdot \mathcal{L}^{trc}_m} \phi^{CTL}_w)$$

Chapter 5

Implementation

For empirical validation purposes, a prototype of our translation algorithm has been implemented in Java. In this section, we present the headlines of this implementation.

As requirements, we wanted this prototype to be independent from both the LSC editor and the model-checker[1] used. Hence, it can be parameterized by the input and output languages chosen. To use a particular editor, the user has to provide the tool with a compiler, that enables the translation program to translate an LSC to its internal representation. Symmetrically, in order to be able to use a model-checker, the user will come up with a generator, i.e. a module designed for writing a syntactically valid proof file for the model-checker used.

5.1 Architecture

The decomposition of the algorithm into four sub-problems induces a natural modular decomposition of the application.

This program is a typical example of the transformational paradigm. It takes an input in a certain form and, through several refinement steps, transforms it into another form. The LSC of figure 5.1 emphasizes the data flow between the different processes involved in the transformation.

Thus, we decided to build classes for the input, the output and the by-products.

The first step of the translation is to build the causal order of the given LSC. From the definition of the causal order $<_m$, definition 1.5, it seems obvious to use directed graphs as its representation. In summary, this step needs an internal representation for LSCs and a class *CausalOrder*, which is a refinement of directed graphs.

The second step builds the trace set of the chart from its causal order. The best representation of the trace set is an acyclic automaton. Hence, we now need a class for handling automata, which can in turn be considered as a specialization of directed graphs. We will need a class *TraceSet*.

In the third step, the algorithm appends two languages. This can be considered as a primitive on automata.

The final step is the generation step. Given a finitely enumerable language and a mapping from LSCs Events to propositional formulae, it will output a proof file suited to a certain type of model-checker. As emphasized above, we did not want to restrict ourselves to a certain kind of formalism and thus, the class *FormulaGenerator* remains abstract.

This architecture is graphically presented as a UML ClassDiagram in figure 5.2. Solid arrows denote an *is-a* relationship between classes of objects while dashed arrows represent a functional dependency between two classes.

The detailed specification of these classes is given in appendix C.

5.2 LSCs Meta-Model

In this section, we present and comment in a detailed way the internal representation of LSCs used by our implementation. The class decomposition of this meta-model is presented as a UML ClassDiagram, in figure 5.3[2].

Every object used in the description of a specification is an *LscObject* and has an identifying name. Some objects, such as charts, locations or messages, have a modal property (temperature). Hence, they are *LscModalObject*, a refinement of *LscObject*.

[1] provided that its logic can express LSCs, for example, CTL*, CTL and LTL.

[2] In figure 5.3, when no role multiplicity is mentioned, (1..1) is the implicit role multiplicity.

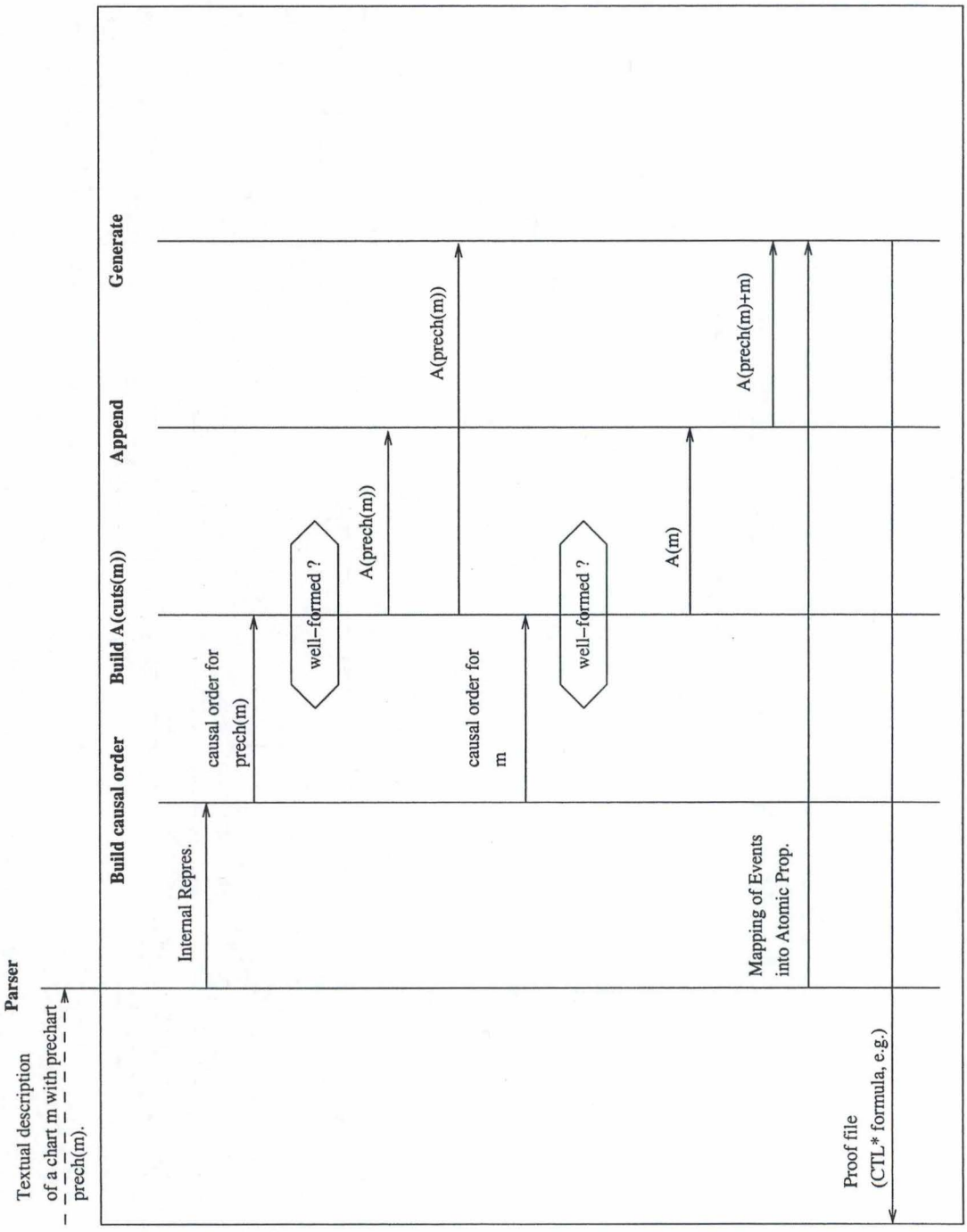


Figure 5.1: Data Flow between translation processes

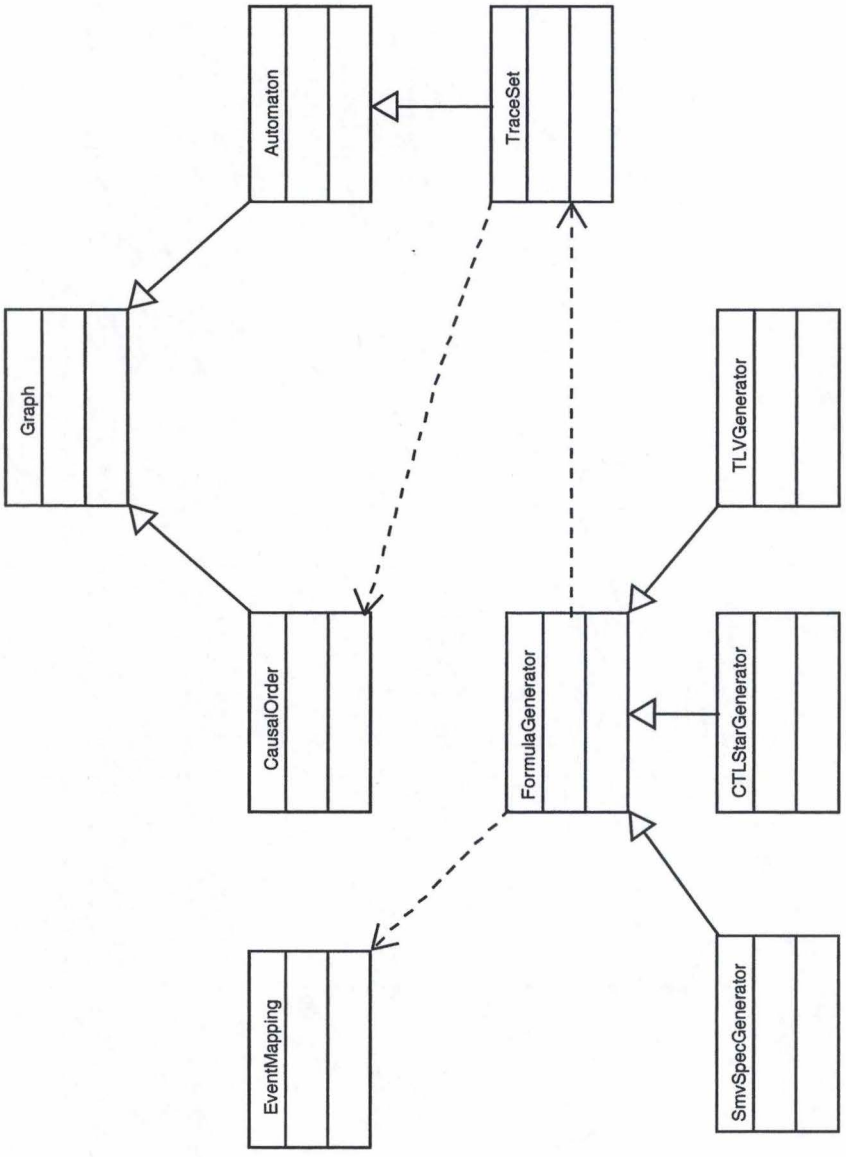


Figure 5.2: Implementation Architecture

An LSC specification is composed of several charts. Every chart is made of two parts : on the one hand, a chart and its activating prechart and, on the other hand, a set of restricted events. We require this set to contain, at least, the events appearing in the chart.

A basic chart consists of instance lines, the locations of which are linked by messages. We represent an instance as a sequence of coregions, each coregion containing at least one location.

Here are the non-graphical constraints that these objects should fulfill:

1. The first coregion of every instance contains exactly one location.
2. The last coregion of every instance contains only cold locations.
3. The messages of a chart only link locations belonging to this chart.
4. Only the initial location of an instance may be not related to an event.
5. If a location is linked to another location by a message, their related events correspond to this message. For example, if a message m is sent on a location l, the event related to l corresponds to the sending of m.

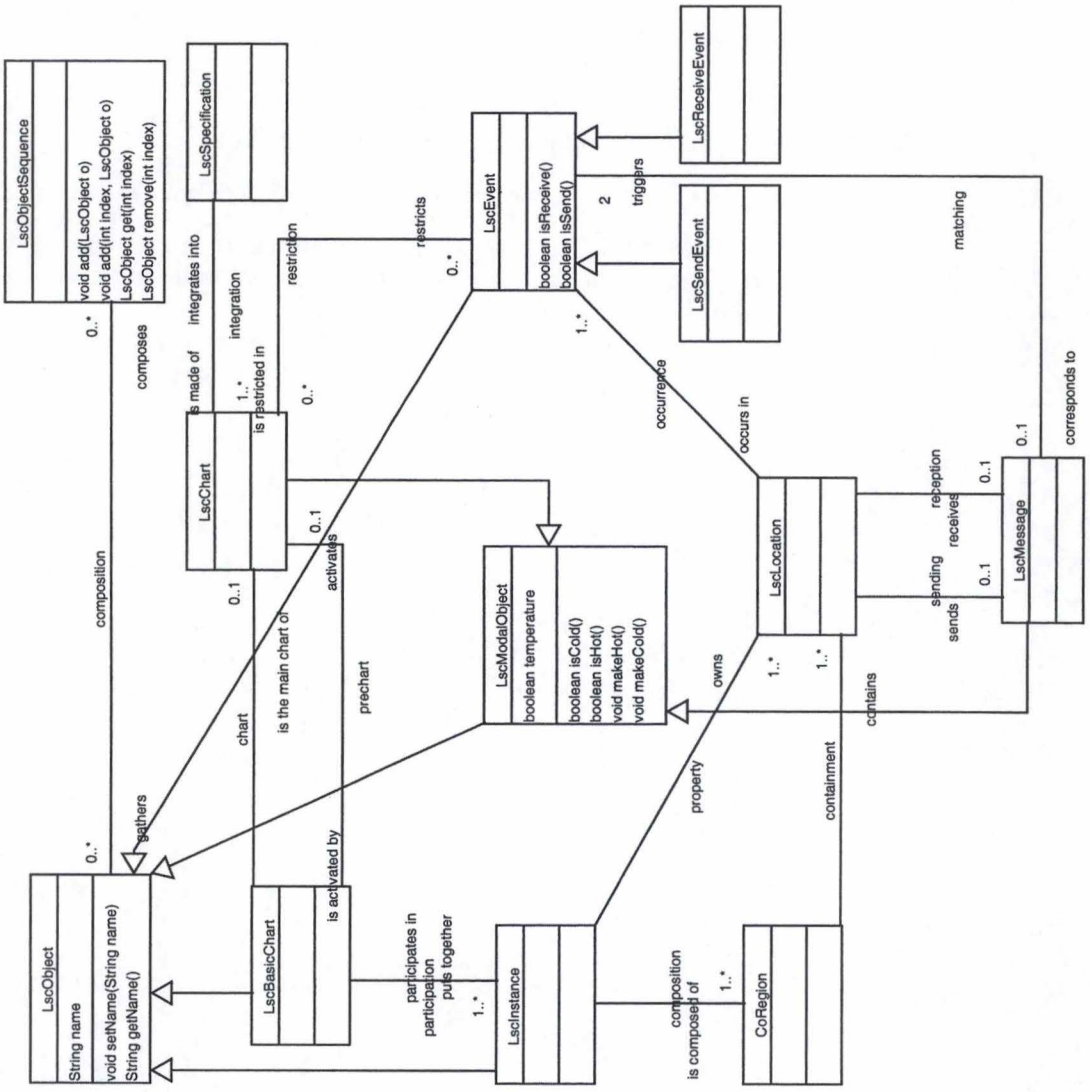


Figure 5.3: LSC Meta-Model

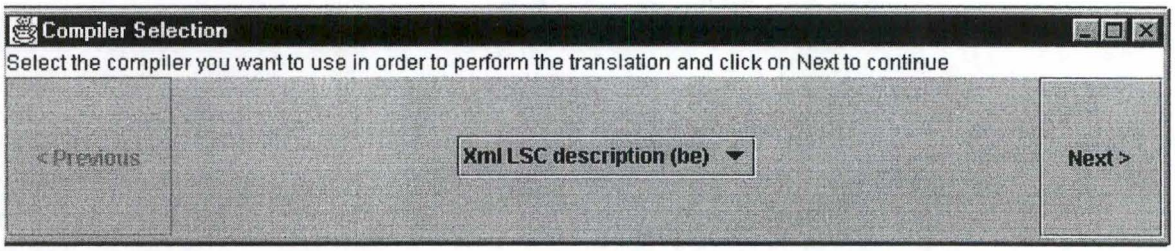


Figure 5.4: GUI's compiler selection

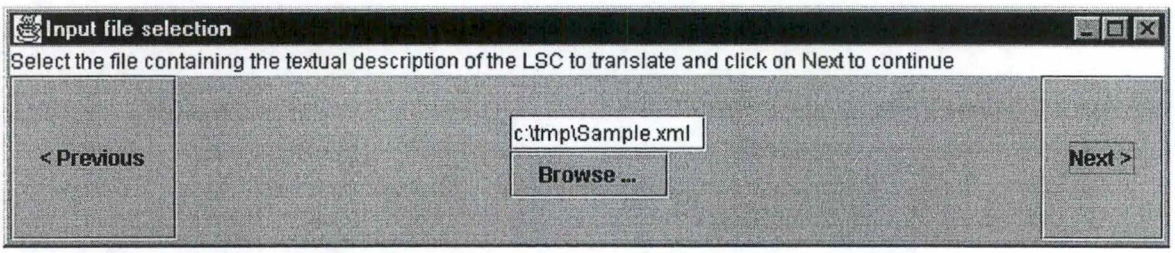


Figure 5.5: GUI's input file selection

5.3 Graphical User Interface

The user interface of our prototype reflects precisely the decomposition already sketched on page 84 and illustrated by an LSC in figure 5.1.

A first window, displayed in figure 5.4, asks the user which compiler he wants to use.

Then, he selects the file containing the textual description of the scenario to check, see figure 5.5.

The user must then provide the translation program with a mapping from the LSC's events to propositional formulae. The window of figure 5.6 is used to perform this task.

In the window of figure 5.7, he chooses the type of model-checker he wants to use. In our example, we only wrote two generators: one for CTL* and one for the CTL SMV model-checker.

Finally, the user gives the program the location of the output file and clicks on "next" to launch the translation process. Errors occurring in the translation are reported in a pop-up window, if any. Otherwise, a window appears to inform the user about the successful termination of the translation.

Figure 5.6: GUI's mapping selection

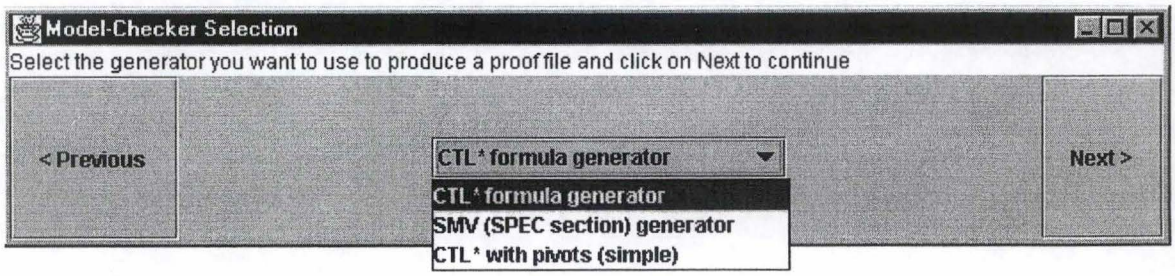


Figure 5.7: GUI's generator selection

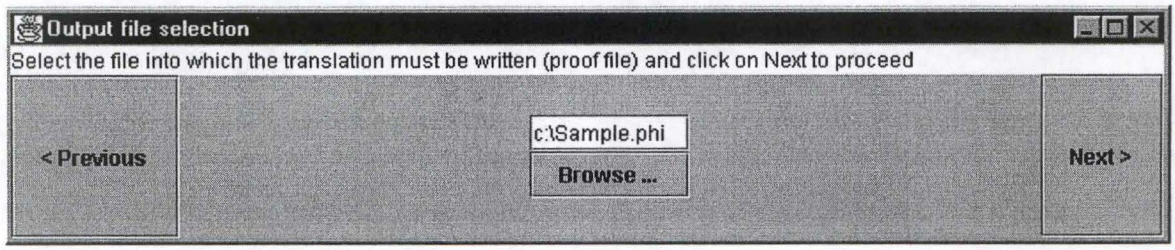


Figure 5.8: GUI's output file selection

5.4 Further development

Two main issues arose when developing this tool prototype.

First, there is a need to completely integrate the translation program into the editor-model-checker chain. The use of this program should be transparent to the user. In his point of view, there should be an editor, in which the inter-objects requirements are managed in use-cases, and from where there is a "one-click" access to the verification.

The user would thus select a scenario, choose the version of the intra-object specification to take into account, possibly give the tool a mapping of inter-object events onto properties over intra-objects variables (i.e. propositional formulae), and then launch the verification. From this moment, the translation would be automatic, the model-checker would verify the satisfaction of the scenario and, if a part of the scenario is left unsatisfied, give the user a natural feedback about it.

This is the second issue that a future implementation should deal with. In order to provide the user with some natural feedback, it seems obvious to use existential LSCs. As a matter of fact, most model-checkers return counter-examples when the formula to be checked was not satisfied. Ideally, such counter-examples, which are traces of the system, could be transformed into existential LSCs.

Then, the tool should be able to superpose the counter-example trace on the scenario that has been violated and to highlight the part of the protocol that differs in the two stories. This non-trivial problem represents, in our point of view, a topic of interest for further work.

Chapter 6

Experimental Results

In this chapter, we report on the results obtained by the verification of the LSCs of section 1.2 against the Ricart-Agrawala system described in section 2.3.1. The system was instantiated to two nodes. We started by analyzing the system assuming that communication was synchronous and then, we relaxed this hypothesis and analyzed a more realistic asynchronous system.

6.1 Synchronous system

At first, we took the hypothesis that the communication in the system was *synchronous*. It meant that no distinction could be made between sending and receiving events, because the communication was infinitely faster than the instance's speed. Or, when an instance sends a message, every instance freezes its progression until the message is received. Therefore, the propositions representing *messages* were considered true when a message reached its destination.

The characteristics of the scenarios under this hypothesis of synchrony are summarized in table 6.1

In table 6.2, we give the results obtained for the validation of the scenarios in a synchronous system, using the symbolic LTL model-checker TLV, [Shahar 00], on a SUN-sparc station Ultra-4 under SunOS release 5.6, with 2176 Mo of RAM.

As explained in section 4.4, we may use CTL*, LTL or CTL model-checkers to verify that a system satisfies a specification expressed in LSC. For convenience, we chose to use TLV, which is developed at Weizmann Institute, Rehovot, Israel, where these tests have been done. Moreover, there are few efficient model-checkers for CTL*.

Because the TLV model-checker uses the SMV language to describe implementation, we were able to use SMV version 2.5.4, [McMillan 00], with very few modifications to the implementation. This CTL model-checker allowed us to compare the efficiency using both techniques (LTL and CTL model-checking).

Our code (for an asynchronous system) is given in appendix B.

The results are displayed in table 6.3. The verification process with this latter model-checker was more efficient.

Name	Type	Size of prechart	Size of chart
No lock-out	Universal	1 message	3 messages
No endless loop	Universal	1 message	1 message
Lock the CS	Universal	2 messages	2 messages
Crossed requests	Existential		3 messages
Crossed requests	No-Story		4 messages
Mutual exclusion	No-Story		2 messages
Long existential	Existential	1 message	9 messages
Long universal	Universal	4 messages	3 messages

Table 6.1: Characteristics of the Ricart-Agrawala's scenarios (in a synchronous system)

6.2 Asynchronous system

If we release the synchronism hypothesis, we are obliged to check every pair of events for every asynchronous message. It results in longer, sometimes multi-trace, formulae. Their characteristics are summarized in table 6.4.

Name	Time	Max. Nbr of BDD Nodes alloc.	Memory allocated
No lock-out	10 min. 39 s.	1,367,361	21.3 Mb
No endless loop	1 min. 40 s.	234,584	4 Mb
Lock the CS	3 min. 23 s.	889,842	14 Mb
Crossed requests (exist.)	50 min. 47s.	1,345,239	21 Mb
Crossed requests (no-story)	14 min. 3 sec.	616,942	9.8 Mb
Mutual exclusion	4 min. 10 sec.	745,744	11.8 Mb
Long existential	6 h. 6 min. 54 sec.	4,519,515	69.4 Mb
Long universal	2 h. 35 min. 58 sec.	4,285,158	65.9 Mb

Table 6.2: Performance of model-checking in LTL (in a synchronous system)

Name	Time	Max. Nbr of BDD Nodes alloc.	Memory allocated
No lock-out	3 min. 21 sec.	40,288	1.8 Mb
No endless loop	1 min. 33 sec.	41,667	1.8 Mb
Lock the CS	1 min. 45 sec.	36,181	1.7 Mb
Crossed requests (exist.)	14 sec.	32,649	1.7 Mb
Crossed requests (no-story)	12 sec.	27,568	1.6 Mb
Mutual exclusion	14 sec.	31,376	1.7 Mb
Long existential	16 sec.	45,084	1.9 Mb

Table 6.3: Performance of model-checking in CTL (in a synchronous system)

The resources needed to verify the scenarios with the TLV model-checker (in LTL) are displayed in table 6.5. Note that the existential charts have been split and that the no-story version of *crossed requests* has only been proven for one of the thirty-six traces.

It takes a lot more time to prove that a property is *not* correct than to prove the validity of a property with similar size. This could be an obstacle to the use of this technique in practice, since we can reasonably assume that it will be used to detect inconsistencies in models rather than to prove that they are correct.

Moreover, as explained in section 4.4.1, in order to prove the satisfaction of an existential chart by a system, in LTL, we check that the system does not violate the chart. In other words, in order to prove a chart m, we check the property $A\neg\beta_m$ and, if the model-checker answers *no*, then we know that the system satisfies m.

Pivoting

As for a synchronous system, we also proved some charts with the CTL model-checker SMV. The resources are shown in table 6.6.

The "*no lock-out*" and "*lock the CS*" formulae have been split, using the first method. The summaries of the resources needed to prove them are in table 6.7 and table 6.8.

Note that, for these two charts, we did not have to use one of the two latter methods, because the first method already yielded optimal formulae (at most 2 events by formula).

It is interesting to note that if, for "*no lock-out*", we obtained a serious gain in efficiency thanks to the splitting (about 40 p.c. less time needed), it took *more time* to verify the split version of "*lock the CS*" than the original formula.

Name	Type	Size of prechart	Size of chart	Nbr of traces
No lock-out	Universal	1 event	6 events	1
No endless loop	Universal	1 event	1 event	1
Lock the CS	Universal	3 events	3 events	2
Crossed requests	Existential		6 events	2
Crossed requests	No-Story		8 events	36
Mutual exclusion	No-Story		2 events	2
Long existential	Existential	1 event	13 events	6

Table 6.4: Characteristics of the Ricart-Agrawala's scenarios (in an asynchronous system)

Name	Time	Max. Nbr of BDD Nodes alloc.	Memory allocated
No lock-out	17 min. 35 sec.	2,735,991	42.2 Mb
No endless loop	2 min. 11 sec.	197,838	3.4 Mb
Lock the CS	17 min. 4 sec.	2,901,035	44.7 Mb
Crossed requests (exist.)	1 h. 54 min. 37 sec.	3,061,678	47.2 Mb
Crossed requests (no-story)	22 min. 33 sec.	3,288,747	50.6 Mb
Mutual exclusion	6 min. 54 sec.	1,179,253	18.4 Mb
Long existential	60 h. 6 min. 21 sec.	23,383,210	357.2 Mb

Table 6.5: Performance of model-checking in LTL (in an asynchronous system)

We explain this by the fact that its prechart is as long as its chart and could not be split. Therefore, we were obliged to check it completely in every split formula. This overhead resulted in this suprizing result. This situation should not appear in real situations, in which the size of the charts should be bigger than what our short example allowed.

Remark that, if it was possible to perform the model-checking of the split formulae in a parallel fashion, the gain would be very high, since it would only have taken 7 minutes to verify the property instead of 17 minutes.

Name	Time (sec.)	Max. Nbr of BDD Nodes alloc.	Memory allocated (bytes)
No lock-out	6 min. 33 sec.	44,877	1.9 Mb
No endless loop	1 min. 57 sec.	45,126	1.9 Mb
Lock the CS	7 min. 28 sec.	44,848	1.9 Mb
Crossed requests (exist.)	26 sec.	72,942	2.3 Mb
Crossed requests (no-story)	16 sec.	32,736	1.7 Mb
Mutual exclusion	18 sec.	37,736	1.9 Mb
Long existential	21 sec.	48,843	1.9 Mb

Table 6.6: Performance of model-checking in CTL (in an asynchronous system)

Name	Time (sec.)	Max. Nbr of BDD Nodes alloc.	Memory allocated (bytes)
φ_0	1 min. 32 sec.	470,337	7.6 Mb
φ_1	2 min. 12 sec.	657,523	10.5 Mb
φ_2	2 min. 51 sec.	708,753	11.3 Mb
φ_3	3 min. 38 sec.	790,982	12.6 Mb
Total	**10 min. 23 sec.**		

Table 6.7: Performance of model-checking in LTL (in an asynchronous system) for *No lock-out* split

Name	Time (sec.)	Max. Nbr of BDD Nodes alloc.	Memory allocated (bytes)
φ_0	5 min. 23 sec.	1,461,442	22.7 Mb
φ_1	6 min. 13 sec.	1,717,690	26.6 Mb
φ_2	7 min. 56 sec.	2,108,504	32.5 Mb
Total	**19 min. 32 sec.**		

Table 6.8: Performance of model-checking in LTL (in an asynchronous system) for *Lock the CS* split

Chapter 7

Conclusion

Our goal was to provide tools to check that intra-object specifications were correct with respect to the user's functional requirements. Scenarios (or *inter-object specifications*) seem to be particularly suitable for describing such requirements.

We advocated the use of *Live Sequence Charts* (LSCs), an extension of Message Sequence Charts (MSCs) [ITU-T 96], to describe functional requirements. We believe that this visual formalism reflects nicely the user's view of his requirements for certain type of application domains, mainly those centered on process or components communication. Distributed and reactive systems are especially the kind of systems that this formalism is suitable to describe.

In this Master thesis, we presented an effective method to translate scenarios, described by LSCs, into temporal logic. This method allows an algorithmic reduction of the requirements' satisfaction problem to the well-known *temporal logic model-checking* problem.

Our algorithm can be considered as naive, since it computes explicitly the semantics of the scenarios. We showed that its complexity was exponential, which turned out to be the time complexity of the translation problem of LSCs into CTL*.

A prototype of a translation application has been developed in Java, in order to demonstrate the effectiveness of our algorithm. It has been validated on scenarios in a distributed system achieving mutual exclusion. This Ricart-Agrawala's system was the running example of our thesis.

The whole verification process has also been empirically validated, on the basis of a distributed system achieving mutual exclusion. These tests showed that the bottleneck of this process was the model-checking problem. However, using efficient tools, such as the CTL model-checker SMV, we could check that an intra-object specification was compliant with an inter-object specification within a couple of minutes.

We also presented a general theory for problem division, called the *pivot theory*. Loosely speaking, we defined a pivot as a particular data that could be used by a program to divide a big problem P into two smaller *independent* problems P_1, P_2 such that solving both P_1, P_2 was equivalent to solving P. Thus, P_1 could be resolved in a parallel fashion with P_2, which accelerates the resolution of the whole problem.

Even if pivot detection is generally not computable, we exhibited two specializations of this concept that appeared to be polynomial-time computable.

This method was used to split a temporal logic formula, equivalent with a scenario, into several smaller formulae, in order to facilitate the verification process.

One could ask if LSCs are really suitable for describing scenarios. Even if it is clear that this language avoids the reduced expressiveness of MSCs, its extended expressiveness increases necessarily its semantics' complexity. It is not obvious anymore that these scenarios are intuitive and readable. In particular, the universality property seems difficult to intuitively understand, as it obliges the specifier to take a vast number of executions (in fact, every possible execution) into account.

Thus, it is important to develop user-friendly means to elicit requirements. Play-in techniques, as described in [Harel 00a], could be an interesting candidate for this. Anyway, we recommend the use of numerous techniques, since requirements engineering is an interactive and iterative process. Requirements should be refined and extended by a round-trip process between the intra-object specification and the inter-object specification, [Bontemps 01].

Moreover, when they are put together into a specification, the executions described by scenarios start to interact with each other, which can cause a consistency problem. Automatically detecting such a problem is not too difficult, using automata-theoretic approaches, for instance. However, it is much more difficult to properly give the user some feedback about this inconsistency.

Another limit of this language comes from the fact that it only aims at describing functional requirements. It introduces a kind of artificial border within requirements, between functional and non-functional requirements. The reality is not that simple. In most cases, some non-functional requirements are strongly related with scenarios that, by nature, describe functional requirements. We particularly think about time properties. Consequently, it would be necessary to extend LSCs in order to take real-time properties into account.

The vision of time that we use in verification is undoubtedly a fundamental issue. The relation between time *running* in the specification and time in the implementation is a complex problem that should be investigated. Should the specification constrain the model of time used in the implementation? Otherwise, how could we link these two different approaches? [Lamport 94b] suggests that logics that are not invariant under stuttering, as our subset of LSCs is, are too expressive to properly reason about programs[1].

In this dissertation, we assumed, thanks to the *hypothesis of mutual exclusion between events* that the scale of time used for LSCs was the same as the model of time used in the intra-object specification, see definition 3.2, hypothesis 2.1 and 3.1. The mutual exclusion hypothesis is strongly related to the semantics of LSCs that has been described as a set of traces, i.e. sequences of observable events, see definition 1.12. This semantics obliged events to occur at distinct moments. In order to define the satisfaction concept, the semantics of an intra-object specification had to reflect this constraint by introducing hypothesis 2.1 that requires that at most one observable event appears in every execution step of the state-machine specification.

Such a requirement seems odd, since it means that the specification constrains the model of time used in the implementation. Of course, this hypothesis is too restrictive.

In some situations, there will be no time running in the implementation between two events, although the specification always imposes events to appear in distinct moments. For instance, consider a transition triggered by the reception of a message and on which another message is sent. Intuitively, there is no delay between the reception of the first message and the sending of the second. Thus, allowing *several* observable events to occur in the same computation step would be intuitively correct. However, this is forbidden by our hypothesis of mutual exclusion between events as it makes it possible for two events to occur in the same state.

We should thus try to release this hypothesis that seems counter-intuitive. Doing this leads to deep modifications of the LSCs semantics. Mainly, an extended semantics should describe the runs of a chart as an infinite sequence of *sets of* propositions rather than an infinite sequence of *propositions*. Further research is needed to address this problem.

In summary, the LSC language is young and will have to grow up. It will only reach its maturity through industrial use and research work. In the verification problem, we believe the pivot theory is the beginning of more work on how to properly *simplify the requirements* to facilitate their verification. In the synthesis problem, much work is still to be done in order to extend the results of [Harel 00b]. Synthesizing scenarios into a readable set of specifications remains a difficult problem.

Last but not least, in this thesis, we presented *techniques* to check that an implementation was compliant with a set of requirements, but we did not say a word about how they should be used. In fact, the automated techniques for relating intra- and inter-object specifications should take place into a wider frame given by a methodology. We miss guidelines to direct the development process. However,

[1] [Lamport 83] is the first paper treating of the importance of invariance under stuttering for specification refinement and implementation relation

such guidelines cannot be given *a priori*. They must be patiently constructed, on the basis of good practices in industry, by experimented and objective reviewers. Experience is the only tool we can use to build such methodologies.

Bibliography

[Aho 00] Alfred Aho, Ravi Sethi & Jeffrey Ullman. Compilateurs. principes, techniques et outils. 2^e cycle, écoles d'ingénieurs. Dunod, Paris, 2000. French version of *Compilers*, pub. Addison-Wesley, ISBN 2-10-005126-1.

[Alur 98] Rajeev Alur & Mihalis Yannakakis. *Model checking of hierarchical state machines.* In Sixth ACM Symposium on the Foundations of Software Engineering, pages 175–188. ACM, 1998.

[Alur 99] Rajeev Alur & Mihalis Yannakakis. *Model Checking of Message Sequence Charts.* In Proceedings of the Tenth International Conference on Concurrency Theory. Springer-Verlag, 1999.

[ArgoUML 98] Project ArgoUML. *ArgoUML*, 1998. The Regents of University of California, Software available on `http://www.argouml.org`.

[Biermann 76] Alan Biermann W. & Ramachandran Krishnaswamy. *Constructing programs from example computations.* IEEE Transactions on Sotware Engineering, vol. 2, pages 141–153, 1976.

[Bodart 94] François Bodart & Yves Pigneur. Conception assistée des systèmes d'information. Méthodes Informatiques et Pratiques des Systèmes. Masson, Paris, 2nd edition, 1994. ISBN 2-225-81807-X.

[Bontemps 01] Yves Bontemps. *An Approach to Round-Trip Requirements Engineering with Statecharts and LSCs*, 2001. Submitted to Requirements Engineering Doctoral Workshop (RE'01). htpp://www.info.fundp.ac.be/~ ybontemp.

[Chan 99] William Chan, Richard Anderson J., Paul Beame, David Jones H., David Notkin & William Warner E. *Decoupling Synchronization from Local Control for Efficient Symbolic Model Checking of Statecharts.* In Proceedings of the 1999 International Conference on Software Engineering (ICSE 99), pages 142–151, Los Angeles, USA, May 1999. ACM.

[Chan 01] William Chan, Richard Anderson J., Paul Beame, David Jones H., David Notkin & William Warner E. *Optimizing Symbolic Model-Checking for Statecharts.* IEEE Transactions on Software Engineering, vol. 27, no. 2, pages 170–190, February 2001. IEECS Log Number 112146.

[Cormen 96] Thomas Cormen H., Charles Leiserson E. & Ronald Rivest L. Introduction to algorithms. The MIT Electrical Engineering and Computer Science Series. McGraw-Hill Book Company and MIT Press, Cambridge, Massachusetts, 1996. 17th printing, ISBN 0-262-03141-8.

[CREWS 99] CREWS. *CREWS: Cooperative Requirements Engineering With Scenarios, ESPRIT Reactive Long Term Research 21.903, http://sunsite.informatik.rwth-aachen.de/CREWS/*, 1996–1999.

[Dams 96] Dennis René Dams. *Abstract interpretation and partition refinement for model-checking.* PhD thesis, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, July 1996. ISBN 90-386-0078-X.

[Davis 93] Alan Davis M. Software requirements. objects, functions and states. Prentice Hall International, New Jersey, 1993. ISBN 0-13-562174-7.

[Derepas 00] Fabrice Derepas, Paul Gastin & David Plainfossé. Avoiding state explosion for distributed systems with timestamps. Submitted for FME 2001, 2000.

[Emerson 90] E. Emerson A. Handbook of theoretical computer science, volume B, chapter 16, Temporal and Modal Logic, pages 997–1072. MIT Press and Elsevier Science Publishers, Cambridge, Massachusetts, 1990. ISBN 0-262-72015-9, J. van Leeuwen.

[Gery 96] Eran Gery & David Harel. *Executable Object Modeling with Statecharts.* In Proc. 18th Int. Conf. Soft. Eng., pages 246–257. IEEE, IEEE Press, March 1996. Revised February 1997. To appear in *IEEE Computer.*

[Harel 98a] David Harel & Werner Damm. *LSCs : Breathing Life into Message Sequence Charts.* Technical Report CS98-09, The Weizmann Institute of Science, Dept of Applied Math. and Comp. Sc., Rehovot, Israel, April 1998. Revised and extended version, September 2000. To appear in *Formal Methods in System Design.*

[Harel 98b] David Harel & Michael Politi. Modeling reactive systems with statecharts: the statemate approach. McGraw-Hill, 1998. ISBN 0-070-26205-5.

[Harel 99] David Harel & Orna Kupferman. *On the behavioral inheritance of State-Based objects.* Technical report, The Weizmann Institute of Science, Dept of Applied Math. and Comp. Sc., Rehovot, Israel, 1999. Submitted as Technical Report MCS99-12.

[Harel 00a] David Harel. *From Play-In Scenarios to Code : An Achievable Dream.* In Proc. Fundamental Approaches to Software Engineering (FASE), volume Vol. 1783 of *Lecture Notes in Computer Science*, pages 22–34. Springer-Verlag, March 2000.

[Harel 00b] David Harel & Hillel Kugler. *Synthesizing State-Based Object Systems from LSC Specifications.* In Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000), Lecture Notes in Computer Science. Springer-Verlag, July 2000. Preliminary Version : August 5, 1999.

[Heymans 98] Patrick Heymans & Eric Dubois. *Scenario-based Techniques for Supporting the Elaboration and the Validation of Formal Requirements.* Requirements Engineering Journal, vol. 3, no. 3–4, pages 202–218, 1998.

[Higham 98] Nicholas Higham J. Handbook of writing for the mathematical sciences. SIAM, Philadelphia, second edition, 1998.

[ITU-T 96] ITU-T. *ITU-T Recommendation Z.120 : Message Sequence Chart (MSC)*, 1996.

[Jackson 99] Michael Jackson A. *Problem Analysis Using Small Problem Frames.* South African Computer Journal, no. 22, pages 47–60, 1999. Special issue on WOFACS'98.

[Jacobson 92] Ivar Jacobson. Object oriented software engineering: a use-case driven approach. ACM Press/Addison-Wesley, 1992.

[Jacobson 99] Ivar Jacobson, James Rumbaugh & Grady Booch. The unified modeling language reference manual. Object Technology Series. Addison-Wesley, 1999. ISBN 0-201-30998-X.

[Jarke 98] M. Jarke, X. T. Bui & J. M. Carroll. *Scenario Management: an Interdisciplinary Approach.* Requirements Engineering Journal, vol. 3, no. 3-4, pages 155–173, 1998.

[Khriss 99] Ismaïl Khriss, Mohammed Elkoutbi & Rudolf Keller K. *Automating the Synthesis of UML Statechart Diagrams from Multiple Collaboration Diagrams.* In Jean Bezivin & Pierre Alain Muller, editors, UML'98: Beyond the Notation, number 1618 in Lecture Notes in Computer Science, pages 132–147. Springer-Verlag, New-York, 1999. Revised version of UML'98 paper. http://www.iro.umontreal.ca/~ keller.

[Knuth 84] Donald Knuth E. The TEXbook, volume a of computers and typesetting. Addison-Wesley, 1984. ISBN 0-201-13448-9.

[Kripke 63] S. Kripke. *A semantical analysis of modal logic I: normal modal propositional calculi.* Zeitschrift für Mathematische Logik und Grundlagen der Mathematik, no. 9, pages 67–96, 1963.

[Krüger 00] Ingolf Heiko Krüger. *Distributed System Design with Message Sequence Charts.* PhD thesis, Institut für Informatik der Technischen Universität München, March 2000. http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2000/krueger.html.

[Kugler 01] Hillel Kugler, David Harel, Amir Pnueli, Lu Yuan & Yves Bontemps. Temporal logic for live sequence charts. Unpublished draft, January 2001.

[Lamport 83] Leslie Lamport. *What Good Is Temporal Logic?* In R. E. A. Mason, editor, Information Processing 83, pages 657–668. Elsevier Publishers, 1983.

[Lamport 94a] Leslie Lamport. LATEX: A document preparation system. Addison-Wesley, Reading, Massachusetts, 1994. second edition, ISBN 0-201-52983-1.

[Lamport 94b] Leslie Lamport. *The Temporal Logic of Actions.* ACM Transactions on Programming Languages and Systems, vol. 16, no. 3, pages 872–923, May 1994.

[Le Charlier 99] Baudouin Le Charlier. *Programmation (deuxième partie) : transparents*, 1999. FUNDP, Namur - Cours de deuxième candidature en Science Economiques et de Gestion (option Informatique).

[Manna 95] Zohar Manna & Amir Pnueli. Temporal verification of reactive systems: Safety. Springer-Verlag, New-York, 1995.

[McMillan 00] K. McMillan L. *The SMV System for SMV version 2.5.4.* Carnegie-Mellon University, November, 6 2000.

[Müller-Olm 99] Markus Müller-Olm, David Schmidt A. & Bernhard Steffe. *Model-Checking, A Tutorial Introduction.* In G. Filé & A. Cortesi, editors, Proc. 6th Static Analysis Symposium, Lecture Notes in Computer Science, New-York, 1999. Springer-Verlag.

[Patterson 98] David Patterson A. & John Hennessy L. Computer organization and design, the hardware/software interface, second edition. Morgan-Kaufmann, San Francisco, California, 1998. ISBN 1-55860-491-X.

[Peled 00] Doron Peled A., Edmund Clarke M. & Orna Grumberg. Model checking. MIT Press, Cambridge, Massachusetts, 2000. ISBN 02-620327-08.

[Pohl 96] Klaus Pohl. Process-centered requirements engineering. Series on Advanced Software Development. Research Studies Press Ltd, 1996. ISBN 0-86380-193-5, American Version publ. by John Wiley & Sons, Inc.

[Schmidt 00] David Schmidt A. Binary relations for abstraction and refinement. Submitted to Elsevier Preprint, November 2000.

[Sedletsky 00] Ekaterina Sedletsky, Amir Pnueli & Mordechai Ben-Ari. *Formal Verification of the Ricart-Agrawala Algorithm.* Technical report, The Weizmann Institute of Science, Dept of Applied Math. and Comp. Sc., Rehovot, Israel, June 2000.

[Selic 94] Bran Selic, Garth Gullekson & Paul Ward T. Real-time object-oriented modeling. Wiley Professional Computing. John Wiley and Sons, Inc., New-York, 1994. ISBN 0-471-59917-4.

[Shahar 00] Elad Shahar. *The TLV Manual.* The Weizmann Institute of Science, Dept of Applied Math. and Comp. Sc., Rehovot, Israel, July 2000.

[Siddiqi 97] J.I. Siddiqi, I.C. Morrey, C.R. Roast & M.B. Ozcan. *Towards Quality Requirements via Animated Formal Specifications.* Annals of Software Engineering, vol. 3, pages 131–155, september 1997.

[Staunstrup 00] Jorgen Staunstrup, Henrik Reif Andersen, Henrik Hulgaard, Jorn Lind-Nielsen, Kim Larsen G., Gerd Behrmann, Kare Kristoffersen, Arne Skou, Henrik Leerberg & Niels Bo Theilgard. *Practical Verification of Embedded Sotware.* IEEE Computer, vol. 33, no. 5, May 2000.

[Systa 00] Tarja Systa, Rudolf Keller K. & Kai Koskimies. *Summary Report of the OOPSLA-2000 Workshop on Scenario-Based Round-Trip Engineering*, October 2000. http://www.cs.uta.fi/~cstasy/oopsla2000/workshop.html.

[Tanenbaum 96] Andrew Tanenbaum S. Computer networks, third edition. Prentice Hall, New Jersey, 1996. ISBN 0-13-394248-1.

[Thomas 90] Wolfgang Thomas. Handbook of theoretical computer science, volume B, chapter 4, Automata on Infinite Objects, pages 134–191. MIT Press and Elsevier Science Publishers, Cambridge, Massachusetts, 1990. ISBN 0-262-72015-9, J. van Leeuwen.

[Vardi 86] Moshe Vardi Y. & Pierre Wolper. *Automata-Theoretic Techniques for Modal Logics of Programs.* Journal of Computer and System Science, vol. 32, no. 2, pages 183–221, April 1986.

[Vardi 98] Moshe Vardi Y. *Automata-Theoretic Approach to Design Verification.* Lecture Notes, 1998. http://www.wisdom.weizmann.ac.il/~vardi/av/index.html.

[Whittle 00] Jon Whittle & Johann Schumann. *Generating Statechart Designs From Scenarios.* In ICSE 2000, Limerick, Ireland, pages 314–323. ACM, 2000.

[Wilhelm 94] Rheinard Wilhelm & Dieter Maurer. Les compilateurs. théorie. construction. génération. Manuels Informatiques. Masson, Paris, 1994. Traduction française, ISBN 2-225-84615-4.

Index

Appendix A

A formal generalization of pivoting

A.1 Pivots : a general definition

Let us first explain the idea of pivoting. We are given a division method, i.e. a method that is able to divide an input into two "smaller" inputs according to some amount of additional information.

For a language, there are some pieces of information that are particularly interesting, the *pivots*. They allow a division of the decision problem of the language into two "easier" problems, using the division method.

Let Σ denote a finite alphabet and Σ^* be the set of finite sequences of letters from Σ.

The division method is a function

$$f : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \Sigma^*$$

Intuitively, $f(w,p) = (w_1, w_2)$ means that w is divided into w_1 and w_2 around p.

We did not fix the exact form of the second parameter since, in some cases, this information will be a letter but sometimes a word or even a language would prove to be useful. We only require that this parameter belongs to a countable set, i.e. such that a bijective function from this set to the natural numbers exists. This is why we wrote $\mathbb{N}$.

Then, for every language $\mathcal{L} \subseteq \Sigma^*$, we want to find what these "interesting pieces of information" are.

Definition A.1 (Pivot) *We say that $p \in \mathbb{N}$ is a* pivot *in $\mathcal{L}$, with respect to f, iff,*

$$\exists\, \mathcal{L}_1, \mathcal{L}_2 :$$

$$\forall w \in \Sigma^* : w \in \mathcal{L}$$
$$\Longleftrightarrow$$
$$f(w,p) = (w_1, w_2) \wedge w_1 \in \mathcal{L}_1 \wedge w_2 \in \mathcal{L}_2$$

Here are some things that we think are remarkable.

First of all, this definition looks quite similar with the *functional reduction*. However, note that here, f is not taken in an *ad hoc* manner, depending on $\mathcal{L}$. Here, f is fixed *a priori*, hence, not depending on $\mathcal{L}$.

Secondly, for a given pivot p, $\mathcal{L}_1$ and $\mathcal{L}_2$ are independent from each other.

Referring to this definition, we call $\mathcal{L}_1, \mathcal{L}_2$ the *pivoted languages* for $\mathcal{L}$ around p.

If p is a pivot in $\mathcal{L}$, then the problem of decision can be distributed among three components : $f, \mathcal{L}_1, \mathcal{L}_2$. Nothing guarantees that the burden of verification is equivalently distributed among these components. Below, we give two examples of division functions that do respectively all the work and absolutely nothing.

Example A.1 (f is workaholic)

$$f(w,p) = \begin{array}{ll} (Y,\varepsilon) & \text{if } w \in p \\ (N,\varepsilon) & \text{if } w \notin p \end{array}$$

So, for every language $\mathcal{L}$, we have that $\mathcal{L}$ is its own and only pivot, with $\mathcal{L}_1 = \{Y\}$ and $\mathcal{L}_2 = \{\varepsilon\}$.

Example A.2 (f is too lazy)

$$f(w,p) = (w,\varepsilon)$$

For every language $\mathcal{L}$, every p is a pivot, with $\mathcal{L}_1 = \mathcal{L}$ and $\mathcal{L}_2 = \{\varepsilon\}$.

In the following definition, we write f_p to denote f where the *second parameter* is fixed to p.

Theorem A.1 (A division method is a reduction function) *For every language $\mathcal{L}$, if p is a pivot in $\mathcal{L}$, then*

$$\left(\begin{array}{c} f_p \text{ is computable} \\ \wedge \mathcal{L}_1, \mathcal{L}_2 \text{ are decidable} \end{array} \right) \Rightarrow \mathcal{L} \text{is decidable}$$

Proof A.1 (Theorem A.1) *The demonstration is straightforward. We just have to prove that we can build an algorithm to decide $\mathcal{L}$.*

1. *Since f_p is computable, we can compute $f(w,p) = (w_1, w_2)$*
2. *Since $\mathcal{L}_1, \mathcal{L}_2$ are decidable, we are able to decide if $w_1 \in \mathcal{L}_1$ and $w_2 \in \mathcal{L}_2$. Therefore, we can check whether or not $w \in \mathcal{L}$, using the definition of a pivot and the assumption that p is a pivot in $\mathcal{L}$.*

□

Here comes a theorem about the decidability of the pivot character. It says, that, even under some strong conditions, in general, determining that a given letter is a pivot in a language is not computable.

Theorem A.2 (Pivots in finite languages are undecidable) *Let $\mathcal{L}$ be a finite language, f be a computable division method. Then, for a given $p \in \mathbb{N}$, determining that p is a f-pivot in $\mathcal{L}$ is* undecidable

Proof A.2 *The proof of this algorithm comes as a consequence of two facts. First of all, the well-known Rice's theorem states that it is impossible to compute virtually any property about functions.*

Theorem A.3 (Rice's theorem) *Let X be a set of functions. Let $\Theta = \{x | f_x \in X\}$ and the characteristic function of Θ, Ξ_Θ, be*

1. *$\Xi_\Theta(x) = true$, if $x \in \Theta$;*
2. *$\Xi_\Theta(x) = false$, if $x \notin \Theta$.*

Then, either

1. *$\Theta = \emptyset$;*
2. *$\Theta = \mathbb{N}$;*
3. *Ξ_Θ is not computable.*

Secondly, as a corollary of definition A.1, if we assume that p is a f-pivot in $\mathcal{L}$, then

$$\begin{array}{c} \forall w \ : \ w \in \mathcal{L} \iff f(w,p) \in \mathcal{L}_1 \times \mathcal{L}_2 \\ \iff \\ \forall w \ : \ w \in \overline{\mathcal{L}} \iff f(w,p) \in \overline{\mathcal{L}_1 \times \mathcal{L}_2} \end{array}$$

Thus, the function $piv(\mathcal{L}, f, p)$, with $\mathcal{L}$ finite, f computable and $p \in \mathbb{N}$, defined as,

1. *true if p is a f-pivot in $\mathcal{L}$;*
2. *false if p is* not *a f-pivot in $\mathcal{L}$.*

should at least check that the given f belongs to

$$F = \{f | \forall w, w \in \overline{\mathcal{L}} \iff f(w,p) \notin \overline{\mathcal{L}_1 \times \mathcal{L}_2}\}$$

F is clearly nontrivial, thus, Rice's theorem applies, and Ξ_F is not computable. Therefore, $piv(\mathcal{L}, f, p)$, is not computable, either.

□

A.1.1 The splitting problem

Remember that we are interested in the *splitting of a temporal logic formula* into several smaller formulae. In this section, we will give a general definition, based on the concept of *pivot*, of what a valid splitting is. Then, we will be able to show that, every time we have a valid splitting, the original formula and the split formulae accept exactly the same runs. In other words, they are equivalent.

We see the initial problem as being composed of two parts : a *generation* function that, given a language in Σ^* returns the corresponding formula (or, equivalently, a language in Σ^ω including all the runs satisfying the formula), and an *abstraction* function that, given an infinite run from Σ^ω, returns the "corresponding" word from Σ^*.

For example, the general form of the CTL* formula equivalent with a chart is a generation function from the trace set of the chart and the abstraction function would project a run onto a word of the trace set, iff the run is valid.

Thus, we have $\mathcal{L} \subseteq \Sigma^*$, $\varphi : 2^{\Sigma^*} \to 2^{\Sigma^\omega}$ and $\alpha : \Sigma^\omega \to \Sigma^*$. Of course, we want the abstraction to be consistent with the generation, i.e. that no run may satisfy the formula if it does not correspond to a word from $\mathcal{L}$ and vice-versa. In the remainder, we write $r \models \varphi(\mathcal{L})$ for $r \in \varphi(\mathcal{L})$.

Hypothesis A.1 (Consistency between generation and abstraction)

$$\forall r \in \Sigma^\omega : r \models \varphi(\mathcal{L}) \iff \alpha(r) \in \mathcal{L}$$

Definition A.2 (Splitting) *A* splitting *is a triple* $(f, \varphi_1/\alpha_1, \varphi_2/\alpha_2)$ *where*

$f : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \Sigma^*$ *is a* division function.

$\varphi_1, \varphi_2 : 2^{\Sigma^*} \to 2^{\Sigma^\omega}$ *are* generation functions.

$\alpha_1, \alpha_2 : \Sigma^\omega \times \mathbb{N} \to \Sigma^*$ *are* abstraction functions. *Their result may depend on the pivot chosen.*

So far, we have just defined the syntactical form of a splitting. Now, we would like to say when a splitting is valid.

Definition A.3 (Valid splitting) *A* valid splitting *is a splitting* $(f, \varphi_1/\alpha_1, \varphi_2/\alpha_2)$ *enjoying the following property :* $\forall \mathcal{L}, \forall p$ *such that* p *is a pivot in* $\mathcal{L}$*, with respect to* f*, and* $\mathcal{L}_1, \mathcal{L}_2$ *are the pivoted languages of* $\mathcal{L}$ *around* p*,*

1. $\forall r \in \Sigma^\omega : r \models \varphi_1(\mathcal{L}_1) \iff \alpha_1(r,p) \in \mathcal{L}_1$
2. $\forall r \in \Sigma^\omega : r \models \varphi_2(\mathcal{L}_2) \iff \alpha_2(r,p) \in \mathcal{L}_2$
3. $\forall r \in \Sigma^\omega : f(\alpha(r),p) = (\alpha_1(r,p), \alpha_2(r,p))$
4. f *is* invertible *for* p*, i.e.*

$$\exists f^{-1} : \forall m, m_1, m_2 : f(m,p) = (m_1, m_2) \iff f^{-1}(m_1, m_2, p) = m$$

The first two properties just state that the generation and abstraction functions are consistent with each other. The third property tells us that the abstraction functions are *symmetric* with the division function. It means that, for a given run, if you abstract the run and then divide it around a pivot, you will get the same two words as if you had abstracted this run with the two abstraction functions provided in the splitting. The last one allows us to rebuild the original word from its pivoted counter-parts, for every pivot.

Theorem A.4 (A valid splitting yields equivalent formulae) *If* $(f, \varphi_1/\alpha_1, \varphi_2/\alpha_2)$ *is a valid splitting, then , for every* $\mathcal{L}$*, for every* p *pivot in* $\mathcal{L}$*, with respect to* f*, yielding the pivoted languages* $\mathcal{L}_1, \mathcal{L}_2$*,*

$$\varphi_1(\mathcal{L}_1) \wedge \varphi_2(\mathcal{L}_2) \iff \varphi(\mathcal{L})$$

Proof A.3 (Theorem A.4) *We first show that*

$$\forall r \in \Sigma^\omega \; : \; r \models \varphi_1(\mathcal{L}_1) \wedge r \models \varphi_2(\mathcal{L}_2) \Rightarrow r \models \varphi(\mathcal{L})$$

We suppose that

$$r \models \varphi_1(\mathcal{L}_1) \quad \wedge \quad r \models \varphi_2(\mathcal{L}_2)$$

holds and we have to prove that

$$r \quad \models \quad \varphi(\mathcal{L})$$

Knowing that $(f, \varphi_1/\alpha_1, \varphi_2/\alpha_2)$ *is a valid splitting, we can use the clauses* (1) *and* (2) *of the definition A.3 to write :*

$$\alpha_1(r,p) \in \mathcal{L}_1 \quad \wedge \quad \alpha_2(r,p) \in \mathcal{L}_2 \tag{A.1}$$

Using (3) *and* (4) *from definition A.3, we get*

$$\alpha(r) \quad = \quad f^{-1}(\alpha_1(r,p), \alpha_2(r,p), p) \tag{A.2}$$

$$f(\alpha(r), p) \quad = \quad (\alpha_1(r,p), \alpha_2(r,p)) \tag{A.3}$$

and since p *is a pivot in* $\mathcal{L}$*, we may deduce from (A.1) and (A.3) that*

$$\alpha(r) \quad \in \quad \mathcal{L}$$

We are thus able to use the consistency hypothesis (hypothesis A.1) to conclude that

$$r \quad \models \quad \varphi(\mathcal{L})$$

□

Now, we show that

$$\forall r \in \Sigma^\omega \; : \; r \models \varphi(\mathcal{L}) \Rightarrow r \models \varphi_1(\mathcal{L}_1) \wedge r \models \varphi_2(\mathcal{L}_2)$$

We make the assumption that

$$r \quad \models \quad \varphi(\mathcal{L}) \tag{A.4}$$

is true and we have to show that

$$r \models \varphi_1(\mathcal{L}_1) \quad \wedge \quad r \models \varphi_2(\mathcal{L}_2)$$

holds, too.

Using the consistency hypothesis (hypothesis A.1), we can rewrite equivalently (A.4) as

$$\alpha(r) \quad \in \quad \mathcal{L} \tag{A.5}$$

Since we are dealing with valid splitting, by the third clause of definition A.3, we have

$$f(\alpha(r), p) \quad = \quad (\alpha_1(r,p), \alpha_2(r,p)) \tag{A.6}$$

.

Provided that p is a pivot, with respect to f, in $\mathcal{L}$, if we put together (A.5) and (A.6), we obtain

$$\alpha_1(r,p) \in \mathcal{L}_1 \quad \wedge \quad \alpha_2(r,p) \in \mathcal{L}_2$$

which, of course, by the first two clauses of definition A.3 is equivalent to

$$r \models \varphi_1(\mathcal{L}_1) \quad \wedge \quad r \models \varphi_2(\mathcal{L}_2)$$

.

□

Appendix B

SMV Implementation of a Ricart-Agrawala System

```
--AR Algorithm

-- VERSION 2.12 (Events stable)

-- cfr Formal Verification of the Ricart-Agrawala Algorithm, 6/8/2000
-- The order of variables is rearranged for minimizing the size
-- of the BDD.

MODULE main
DEFINE

-- Nbr of nodes
NPROCS := 2;

-- Upper bound for the timestamp
MAX_NUM := 4;

VAR

-- Reply channels from 0_1

 partp[1][2] : process pcomp(chp[1][2],count[2],2,1,msg_snd,msg_rcv,stamp);


-- Reply channels from 0_2

 partp[2][1] : process pcomp(chp[2][1],count[1],1,2,msg_snd,msg_rcv,stamp);


-- Reply channels from 0_3



-- Request channels for 0_1
```

```
 partq[2][1] : process qcomp(1,2, chp[1][2], chq[2][1], defrep[1][2],
                              number[1], max[1], requesting[1],
                              msg_snd,msg_rcv,stamp);




-- Request channels for 0_2

 partq[1][2] : process qcomp(2,1, chp[2][1], chq[1][2], defrep[2][1],
                              number[2], max[2], requesting[2],
                              msg_snd,msg_rcv,stamp);



-- Processes that avoid timestamp to go over MAX_NUM

reducer[1] : process reduce_gaps(1,chq,number,max,msg_snd,msg_rcv,stamp);
reducer[2] : process reduce_gaps(2,chq,number,max,msg_snd,msg_rcv,stamp);
reducer[3] : process reduce_gaps(3,chq,number,max,msg_snd,msg_rcv,stamp);
reducer[4] : process reduce_gaps(4,chq,number,max,msg_snd,msg_rcv,stamp);
--reducer[5] : process reduce_gaps(5,chq,number,max);
--reducer[6] : process reduce_gaps(6,chq,number,max);


-- Represents the occurrence of events in the system

msg_snd : 0..8;
msg_rcv : 0..6;

-- msg_snd = 0 <=> Null
-- msg_snd = 1 <=> (Node1,Node2.request) sent
-- msg_snd = 2 <=> (Node2,Node1.request) sent
-- msg_snd = 3 <=> (Node1,Node2.reply)   sent
-- msg_snd = 4 <=> (Node2,Node1.reply)   sent
-- msg_snd = 5 <=> (Node1,CS.enter)
-- msg_snd = 6 <=> (Node2,CS.enter)
-- msg_snd = 7 <=> (Node1,CS.exit)
-- msg_snd = 8 <=> (Node2,CS.exit)


-- msg_rcv = 0  <=> Null
-- msg_rcv = 1  <=> (env,Node1.ASK_CS)
-- msg_rcv = 2  <=> (env,Node2.ASK_CS)
-- msg_rcv = 3  <=> (Node1,Node2.request) received
-- msg_rcv = 4  <=> (Node2,Node1.request) received
-- msg_rcv = 5  <=> (Node1,Node2.reply)   received
-- msg_rcv = 6  <=> (Node2,Node1.reply)   received
```

```
-- Redundant
chq[2][2] : 0..MAX_NUM;
chq[1][1] : 0..MAX_NUM;

chp[1][1] : boolean;

defrep[1][1] : boolean;
chp[2][2] : boolean;
defrep[2][2] : boolean;

-- for debugging purpose only
stamp: {main,p,q,reduce,str};

-- The first main process (O_1)
 O[1] : process mcomp(chp,chq,defrep,1,p[1],count[1],
                number[1],max[1],requesting[1],NPROCS,MAX_NUM,msg_snd,msg_rcv,stamp);

-- counter for the loops in O_1
p[1] : 0..NPROCS+1;

-- counter for the number of replies received by O_1
count[1] : 0..NPROCS - 1;



-- The second main process (O_2)
 O[2] : process mcomp(chp,chq,defrep,2,p[2],count[2],
                number[2],max[2],requesting[2],NPROCS,MAX_NUM,msg_snd,msg_rcv,stamp);

-- counter for the loops in O_2
p[2] : 0..NPROCS+1;

-- true iff O_2 requests the entry into the CS
requesting[2] : boolean;

-- true iff O_1 requests the entry into the CS
requesting[1] : boolean;

-- counter for the number of replies received by O_2
count[2] : 0..NPROCS - 1;

-- defferred replies from O_2 to O_1
defrep[2][1] : boolean;

-- Channel for replies from O_2 to O_1
chp[2][1] : boolean;

-- osn in O_2
number[2] : 0..MAX_NUM;

-- channel for queries from O_1 to O_2
chq[1][2] : 0..MAX_NUM;
```

```
-- hsn in O_2
max[2] : 0..MAX_NUM;

-- osn in O_1
number[1] : 0..MAX_NUM; --3 bits

-- channel for queries from O_2 to O_1
chq[2][1] : 0..MAX_NUM;

-- hsn in O_1
max[1] : 0..MAX_NUM;

-- Channel for replies from O_1 to O_2
chp[1][2] : boolean;

-- defferred replies from O_1 to O_2
defrep[1][2] : boolean;

ASSIGN

init(stamp) := str;

init(count[1]) := 0;
init(count[2]) := 0;

init(p[1]):=0;
init(p[2]):=0;

init(number[1]) := 0;
init(number[2]) := 0;


init(max[1]) := 0;
init(max[2]) := 0;


init(requesting[1]):=0;
init(requesting[2]):=0;

init(chp[1][2]):=0;
init(chp[2][1]):=0;

init(chq[1][2]):=0;
init(chq[2][1]):=0;

init(defrep[1][2]):=0;
init(defrep[2][1]):=0;


init(msg_snd) := 0;
init(msg_rcv) := 0;

next(stamp) := str;
next(msg_snd) := 0;
```

```
next(msg_rcv) := 0;

JUSTICE

   ! O[1].pc = m5, ! O[1].pc = m2,  ! O[1].pc = m31, ! O[1].pc = m32,
   ! O[1].pc = m6, ! O[1].pc = m71, ! O[1].pc = m72,

   ! O[2].pc = m5, ! O[2].pc = m2,  ! O[2].pc = m31, ! O[2].pc = m32,
   ! O[2].pc = m6, ! O[2].pc = m71, ! O[2].pc = m72,

   ! (O[1].pc = m4 & count[1] = 0), ! (O[2].pc = m4 & count[2] = 0),

   chp[1][2] = 0, chp[2][1] = 0, chq[1][2] = 0, chq[2][1] = 0



MODULE mcomp(ch_p, ch_q, def_rep,act, p1,c, osn, hsn, req,bound, bnum,
             msg_snd,msg_rcv,stamp)

-- Represents the behaviour of the Main component of a node

-- ch_p is the communication channel for the replies
--(where ch_p[i][j] means i sends a reply to j)
-- ch_q is the communication channel for the requests
--(where ch_q[i][j] means i sends a request to j)
-- def_rep is the list of defferred replies
--(where def_rep[i][j] means i should send a reply to j, ASAP)
-- act is the actual index of this instance
-- p1 is a counter for the loops (for every Main process do ... )
-- c is the counter for the received replies
-- osn is the Own Sequence Number of this instance (current timestamp)
-- hsn is the Highest Sequence Number received by this instance
-- req is true iff this instance is requesting for an entry into the CS
-- bound is the number of nodes existing in the system (for the loops)
-- bnum is the upper bound for the seq number
-- msg_snd is the sending event occurring at this  state
-- msg_rcv is the receive event occurring at this state
-- stamp is for debugging purpose only

VAR

-- the p-counter, indicates in which state of the algorithm this instance is.
pc: {m1,m2,m31,m32,m4,m5,m6,m71,m72};  -- 4 bits


ASSIGN

next(stamp) := main;

-- Receive events in the main component
-- Only the ASK_CS, sent from the environment.
next(msg_rcv) :=
case
pc = m1 & next(pc) = m2 & act = 1 : 1;
```

```
pc = m1 & next(pc) = m2 & act = 2 : 2;
1 : 0 ;
esac;

init(pc) := m1;

-- for doc about this, cfr Formal Verification of the Ricart-Agrawala Algorithm
next(pc) := case
   pc=m1 : {m1,m2};
   pc=m2 & hsn < bnum: m31;
   pc=m31 & p1<= bound : m32;
   pc=m31 & p1 > bound : m4;
   pc=m32 : m31;
   pc = m4 & c = 0: m5;
   pc =m4 & c>0: m4;
   pc = m5 : m6;  -- critical
   pc = m6 : m71;
   pc = m71 & p1<= bound : m72;
   pc = m71 & p1 > bound : m1;
   pc=m72 : m71;
   1 : pc;
  esac;

-- We determine the event thanks to the value
-- of the P-Counter
next(msg_snd) := case
pc = m32 & next(pc) = m31 & ! act = p1 & act = 1 : 1;  -- 1 sends a request to 2
pc = m32 & next(pc) = m31 & ! act = p1 & act = 2 : 2;  -- 2 sends a request to 1
pc = m4 & next(pc) = m5 & act = 1 : 5 ; -- 1 enters CS
pc = m4 & next(pc) = m5 & act = 2 : 6 ; -- 2 enters CS
pc = m5 & next(pc) = m6 & act = 1 : 7 ; -- 1 exits CS
pc = m5 & next(pc) = m6 & act = 2 : 8 ; -- 2 exits CS
pc = m72 & def_rep[act][2] & act = 1  : 3; -- 1 sends a reply to 2
pc = m72 & def_rep[act][1] & act = 2  : 4; -- 2 sends a reply to 1
1 : 0 ;
esac;



-- be requesting if between the second and 6th step.
next(req) :=
  case
   pc = m2 & hsn < bnum : 1;
   pc = m6 : 0;
   1 : req;
  esac;

-- if necessary (in the second step), choose a new timestamp.
next(osn) :=
  case
   pc = m2 & hsn < bnum : hsn+1;
   1 : osn;
  esac;
```

```
next(c) :=
  case
    pc = m2 & hsn < bnum : 1; -- (NPROCS - 1);
    1 : c;
  esac;

-- the index for the loops.
-- Initialise it before a loop and increase it within a loop.
next(p1) :=
  case
    pc = m2 & hsn < bnum : 1;
    (pc = m32 | pc = m72) & p1<=bound: p1 +1;
    pc = m6: 1;
    1 : p1;
  esac;


-- When to post a request from this instance to instance 1
next(ch_q[act][1]) :=
  case
    pc = m32 & p1 = 1 & !(act=p1)  : osn;
--  if in first step of the request loop
--and not to itself, post
    1 : ch_q[act][1];
  esac;

-- When to post a request from this instance to instance 2
next(ch_q[act][2]) :=
  case
    pc = m32 & p1 = 2 & !(act=p1)  : osn;
--  if in second step of the request
--  and not to itself, loop
    1 : ch_q[act][2];
  esac;

-- When to post a reply from this instance to instance 1
next(ch_p[act][1]) :=
  case
    pc = m72 & def_rep[act][1] : 1;
--  if in first step of the reply loop
--  and reply has been defferred, post
    1 : ch_p[act][1];
  esac;

-- When to post a reply from this instance to instance 2
next(ch_p[act][2]) :=
 case
    pc = m72 & def_rep[act][2] : 1;
--  if in second step of the reply loop
--  and reply has been defferred, post
    1 : ch_p[act][2];
  esac;

-- There is no need to consider a reply
```

```
-- as deferred since it has been sent in step 7.2
next(def_rep[act][1]) :=
  case
    pc = m72 & def_rep[act][1] : 0;
    1 : def_rep[act][1];
  esac;

-- There is no need to consider a reply
-- as deferred since it has been sent in step 7.2
next(def_rep[act][2]) :=
  case
    pc = m72 & def_rep[act][2] : 0;
    1 : def_rep[act][2];
  esac;



-- Handling the Requests

MODULE qcomp(act,nbr,ch_p,ch_q,def_rep,osn,hsn,req,msg_snd,msg_rcv,stamp)

-- act the actual number of the node this instance belongs to
-- nbr the number of the instance from which the request comes.
-- ch_p the channel for transporting replies
-- ch_q the channel for transporting requests
-- def_rep the list of defferred replies.
-- osn the current timestamp for this node
-- hsn the highest timestamp received yet.
-- req true iff the node is requesting an entry into the CS
-- msg_snd the send event occurring in this state
-- msg_rcv the receive event occurring in this state
-- stamp for debugging purpose only

ASSIGN

next(stamp) := q;

-- Q-Comp can receive requests.  It has to determine the sender and receiver.
next(msg_rcv) :=
case
ch_q > 0 & act = 1 & nbr = 2 : 4; -- 1 receives a request from 2
ch_q > 0 & act = 2 & nbr = 1 : 3; -- 2 receives a request from 1
1 : 0;
esac;

-- Q-Comp can send replies
next(msg_snd) :=
case
(ch_q>0) & (ch_q<osn | (ch_q=osn & nbr<act) | !req)
 & act = 1 & nbr = 2 : 3; -- 1 sends a reply to 2
(ch_q>0) & (ch_q<osn | (ch_q=osn & nbr<act) | !req)
 & act = 2 & nbr = 1 : 4; -- 2 sends a reply to 1
1 : 0;
esac;
```

```
-- Always senses the request channel. Empties it after reading.
next(ch_q) := 0;

next(hsn) :=
  case
    (ch_q > 0) & (hsn < ch_q) : ch_q;
     1 : hsn;
  esac;

-- When should the Q-Comp post a reply ?
next(ch_p) :=
  case
    (ch_q>0) & (ch_q<osn | (ch_q=osn & nbr<act) | !req): 1;
-- When there is a query such that (osnq,idq)<(osn,own_id) (older, cfr article)
-- or the node is not requesting (interested in the CS)
     1 : ch_p;
  esac;

-- When should a reply be defferred ?
next(def_rep) :=
  case
    (ch_q>0) &  !(ch_q<osn | (ch_q=osn & nbr<act) | !req): 1;
-- When there is a query such that
-- it is not older than  the query sent by the node
-- and the node is requesting (interested in the CS)
    1 : def_rep;
  esac;


-- Handling the reply channel

MODULE pcomp(ch_p,c,act,from,msg_snd,msg_rcv,stamp)

-- ch_p the channel transporting the replies
-- c the number of replies to be received by this node
-- act the actual number of the instance node to which this process belongs
-- from the number of the instance node
--  that sends the reply on the channel this process is watching
-- msg_snd the sending event occurring at this stage
-- msg_rcv the receive event occurring at this stage
-- stamp for debugging purpose only

ASSIGN

next(stamp) := p;


-- P-Comp doesn't send anything
next(msg_snd) := 0;

-- Decides, if a reply has been received, which kind of reply it is.
-- 0_1 --> 0_2 or 0_2 --> 0_1
next(msg_rcv) :=
```

```
case
ch_p>0 & c>0 & act = 1 & from = 2 : 6;
ch_p>0 & c>0 & act = 2 & from = 1 : 5;
1 : 0;
esac;

-- Always senses the reply channel.  Empties it after reading.
next(ch_p) := 0;

-- When should we consider that this node has received a new reply ?
next(c) :=
  case
-- When there is a reply in the channel
-- and the node is waiting for more replies.
   ch_p>0 & c > 0 : c -1;
   1 : c;
  esac;


MODULE reduce_gaps(slot,chq,number,max,msg_snd,msg_rcv,stamp)

-- This process tries to keep the sequence numbers
-- as small as possible, in order to
-- avoid state explosion due to timestamps.
DEFINE
  possible := chq[1][2] !=slot & chq[2][1] !=slot &
              number[1]!=slot & number[2]!=slot &
              max[1]!=slot & max[2]!=slot;

ASSIGN

  next(stamp) := reduce;

  next(msg_snd) := 0;
  next(msg_rcv) := 0;

  next(chq[1][2]):=case
     possible & chq[1][2] > slot : chq[1][2] - 1;
     1 : chq[1][2];
   esac;

  next(chq[2][1]):= case
    possible & chq[2][1] > slot : chq[2][1] - 1;
    1 : chq[2][1];
   esac;

  next(number[1]):=case
    possible & number[1]>slot : number[1]-1;
    1 : number[1];
   esac;

  next(number[2]):=case
    possible & number[2]>slot : number[2]-1;
    1 : number[2];
```

```
esac;

next(max[1]):=case
  possible & max[1]>slot : max[1]-1;
  1 : max[1];
  esac;

next(max[2]):=case
  possible & max[2]>slot : max[2]-1;
  1 : max[2];
esac;
```

Appendix C

Translation Algorithm : Detailed Specification of the Java Implementation

This documentation has been generated by Sun's Javadoc tool (in JDK 1.3), using a doclet for LaTeX by Gregg Wonderly - C2 Technologies Inc.

Contents

Chapter 1

Package be.ac.fundp.info.albert.lsc.graph.automato

Package Contents *Page*

Classes

1.1 Classes

1.1.1 Class **Automaton**

A class for representing generic automata (NFA,DFA,ADFA,ANFA, min NFA, min DFA).

This class provides methods to determine the class of automata this automaton belongs to and to transform an automaton from one class to the other (not implemented yet).

This class sets no restriction on the number of states, as well as on the number of transitions in the automaton. Since it uses generic States and generic Transitions, the form of the State and the Labels is not restricted, either.

Declaration

public class Automaton
extends be.ac.fundp.info.albert.lsc.graph.Graph

Constructors

- *Automaton*
  `public` **Automaton**`( )`
  - **Usage**
    - Creates an empty Automaton.

- *Automaton*
  `public` **Automaton**`( java.util.Collection` **states**`, java.util.Collection` **initStates** `)`
  - **Usage**
    - Creates a new Automaton with the collection of States states, out of which initStates is a collection of initial states and no final states. Thus, the language of the constructed Automaton is empty.
  - **Parameters**
    - `states` - the collection of states to add to this automaton.
    - `initStates` - the states in c that are initial.

- *Automaton*
  `public` **Automaton**`( java.util.Collection` **states**`, java.util.Collection` **initStates**`, java.util.Collection` **finalStates** `)`
  - **Usage**
    - Creates a new Automaton with the collection of States states, out of which initStates is a collection of initial states and finalStates is a collection of final states.
  - **Parameters**
    - `states` - the collection of states to add to this automaton.

  * `initStates` - the states in states that are initial.
  * `finalStates` - the states in states that are accepting.

- *Automaton*
  `public` **Automaton(** `be.ac.fundp.info.albert.lsc.graph.automaton.State` **init)**

  - **Usage**
    * Creates a new Automaton with only one State (initial).
  - **Parameters**
    * `init` - the initial state of the new automaton.

METHODS

- *addState*
  `public void` **addState(** `be.ac.fundp.info.albert.lsc.graph.automaton.State` **s)**

  - **Usage**
    * adds a State s to this automaton, if it does not already belong to its set of states.
  - **Parameters**
    * `s` - the State to add to this automaton.

- *appendTo*
  `public void` **appendTo(** `be.ac.fundp.info.albert.lsc.graph.automaton.Automaton` **prefix)**

  - **Usage**
    * Appends this Automaton to the given prefix Automaton. It results in modifying this Automaton such that its language corresponds to the concatenation of the language of prefix and the language of this. Warning, if prefix and this share any common state, they will not be merged or the result will be totally undetermined.

      prefix remains unchanged after this method finishes.
  - **Parameters**
    * `prefix` - the Automaton this will be appended to.
  - **Returns** - none. This method works by side-effect on this Automaton.

- *containsState*
  `public boolean` **containsState(**
  `be.ac.fundp.info.albert.lsc.graph.automaton.State` **s)**

  - **Usage**
    * tests if this automaton contains the given State.
  - **Parameters**
    * `s` - the State to look for in this Automaton.

- *createFromText*
  `public static Automaton` **createFromText(** `java.io.BufferedReader` **txt)**

- **Usage**
  - Creates an Automaton from a textual description.

    The syntax of the file for a textual description is the following :
    Automaton ::= INIT initsect REJECTING rejectsect ACCEPTING acceptsect
    [TRANS transsect] END CRLF
    initsect ::= CRLF (idstate CRLF)*
    rejectsect ::= CRLF (idstate CRLF)*
    acceptsect ::= CRLF (idstate CRLF)*
    transsect ::= CRLF (idstate CRLF label CRLF idstate CRLF)*
    idstate ::= char+
    label ::= char+

    An automaton is defined as a set of initial states, defined in the "INIT" section, a set of accepting states, defined in the "ACCEPTING" section and a set of non-accepting (and non-initial, either) states, defined in the "REJECTING" section. Every state is defined by giving its identifier. Note that these three sets are disjunct. Intersecting them gives the set of all the states in the automaton. Between two states, a transition can be defined in the "TRANS" section. A transition is defined by giving, on a first line, its origin state, on a second line, its label and on a third line, its destination state. Note that CRLF denotes the end-of-line character.
- **Parameters**
  - `txt` - a BufferedReader from which the textual description of the automaton to be created will be read.
- **Returns** - The automaton described in txt.
- **Exceptions**
  - `java.io.IOException` - if anything bad occurs while reading txt.
  - `be.ac.fundp.info.albert.lsc.graph.automaton.BadAutomatonException` - If the textual description contained in txt is not correct with respect to the defined syntax.

- *getAcceptingStates*
  `public Collection` **getAcceptingStates( )**
  - **Usage**
    - Gets a copy of the final states of this automaton.
  - **Returns** - a copy of the set of final (accepting) states of this automaton.

- *getInitialStates*
  `public Collection` **getInitialStates( )**
  - **Usage**
    - Gets a copy of the initial states of this automaton.
  - **Returns** - a copy of the set of initial states of this automaton.

- *getShortestAcceptingPath*
  `public Vector` **getShortestAcceptingPath( )**

- *getState*
  `public State` **getState**`( be.ac.fundp.info.albert.lsc.graph.automaton.State s )`

  - **Parameters**
    - `s` - the State to retrieve in this automaton.
  - **Returns** - the State in this automaton identical to s, if any. null, otherwise.

- *getWords*
  `public Collection` **getWords**`( )`

  - **Usage**
    - Fetches all the words of the language described by this automaton. In order to be able to perform this operation, the language must of course be finite. In terms of automata, it means that this automaton must be acyclic. If it is not, a BadAutomatonException will be thrown.
  - **Returns** - a Collection containing all the words (i.e. Vectors of letters) in the finite language described by this Automaton. There is no repetition in the Collection, i.e. no word appears twice in it.
  - **Exceptions**
    - `be.ac.fundp.info.albert.lsc.graph.automaton.BadAutomatonException` - iff this Automaton is not acyclic, and thus its language is not finitely enumerable.

- *isAcyclic*
  `public boolean` **isAcyclic**`( )`

  - **Usage**
    - Checks whether or not this Automaton belongs to the class of Acyclic automata and thus, describes a finite language.
  - **Returns** - true iff this Automaton is acyclic.

- *isDeterministic*
  `public boolean` **isDeterministic**`( )`

  - **Usage**
    - Checks whether or not this Automaton belongs to the class of Deterministic automata. As a remainder, a deterministic automaton is an automaton such that : 1) it contains one and only one initial state 2) it contains no null (epsilon in automata-theoretic terms) transition. 3) in every State, for every label of the input alphabet there is at most one transition starting from this State.
  - **Returns** - true iff this Automaton is deterministic.

- *isEmptyLanguage*
  `public boolean` **isEmptyLanguage**`( )`

  - **Usage**
    - Checks whether or not the language of this Automaton is empty. Remember that the empty is language is different from the language composed only of the null (epsilon in automata-theoretic terms) transition.

      An automaton has an empty language iff there is no accepting path in it. An accepting path is a path from one initial state to one final (accepting) state.

– **Returns** - true iff the language of this Automaton is empty.

- *main*
  `public static final void` **main**`( java.lang.String [] argv )`
  - **Usage**
    - Creates an Automaton from the textual description stored in the file given as argument. Gets the first possible cycle of this Automaton. Decides if this Automaton has an empty language, is Acyclic or Deterministic. Computes the left-most path in this Automaton. During this execution, prints out the resource usage.

- *makeAccepting*
  `public void` **makeAccepting**`( java.util.Collection c )`
  - **Usage**
    - makes a Collection of States accepting.
  - **Parameters**
    - `c` - the collection of states to make accepting.

- *makeAccepting*
  `public void` **makeAccepting**`(`
  `be.ac.fundp.info.albert.lsc.graph.automaton.State s )`
  - **Usage**
    - makes a State accepting, if it belongs to this Automaton.
  - **Parameters**
    - `s` - the State to make accepting.

- *makeInitial*
  `public void` **makeInitial**`( java.util.Collection c )`
  - **Usage**
    - makes a Collection of States initial.
  - **Parameters**
    - `c` - the collection of states to make initial.

- *makeInitial*
  `public void` **makeInitial**`( be.ac.fundp.info.albert.lsc.graph.automaton.State s`
  `)`
  - **Usage**
    - makes a State initial. Adds it to the set of initial states.
  - **Parameters**
    - `s` - the state to make initial.

- *makeNotInitial*
  `public void` **makeNotInitial**`( java.util.Collection c )`
  - **Usage**
    - makes a Collection of States not initial.

- **Parameters**
  - `c` - the collection of states to make not initial.

- *makeNotInitial*
  `public void` **makeNotInitial(**
  `be.ac.fundp.info.albert.lsc.graph.automaton.State s )`
  - **Usage**
    - makes a State not initial. Removes it from the set of initial states.
  - **Parameters**
    - `s` - the state to make initial.

- *makeRejecting*
  `public void` **makeRejecting(** `java.util.Collection c )`
  - **Usage**
    - makes a Collection of States rejecting.
  - **Parameters**
    - `c` - the collection of states to make rejecting.

- *makeRejecting*
  `public void` **makeRejecting(**
  `be.ac.fundp.info.albert.lsc.graph.automaton.State s )`
  - **Usage**
    - makes a State rejecting (i.e. NOT accepting).
  - **Parameters**
    - `s` - the State to make rejecting.

- *pathToWord*
  `public static final Vector` **pathToWord(** `java.util.Vector path )`
  - **Usage**
    - Transforms a path into a word.
      A path is a Vector of Transition.
      A word is a Vector of labels of Transitions.
  - **Parameters**
    - `path` - the path to transform into word.
  - **Returns** - Given a path (t1,...,tn), a Vector (l1,...,ln), where, for every i:1<=i<=n: li is the label of ti.

- *removeState*
  `public void` **removeState(** `be.ac.fundp.info.albert.lsc.graph.automaton.State s )`
  - **Usage**
    - removes a State from this automaton.
  - **Parameters**
    - `s` - the State to remove from this automaton.

- *toString*
  `public String` **`toString( )`**
  - **Usage**
    - Format an Automaton into a displayable form.
  - **Returns** - a String containing a human-readable form of this automaton.

1.1.2 Class BadAutomatonException

This exception is thrown by instances of class Automaton whenever the precondition to a method is not fulfilled.

Declaration

public class BadAutomatonException
extends java.lang.Exception

Constructors

- *BadAutomatonException*
  `public` **`BadAutomatonException( )`**
- *BadAutomatonException*
  `public` **`BadAutomatonException(`** `java.lang.String` **`msg )`**

1.1.3 Class DummyState

A Dummy State. A Dummy State is used to perform some paths properties in automata, in combination with null-transitions. There are two kinds of interesting dummy states : initial dummy states, that can represent any initial state, and final dummy states, that can represent any accepting state, when properly used.

An initial or final dummy state is only equals to another initial (resp. final) dummy state.

An initial dummy state is always smaller than every other state, while a final dummy state is always bigger.

Note that the concepts of initial and final, as they are used in this context, are totally independent from the "position" of the State in the Automaton. It is only the very nature of the State. Nothing keeps a user from creating a Dummy Initial State and then to use it as a regular, rejecting, non-initial State, in a given Automaton.

Declaration

public class DummyState
extends be.ac.fundp.info.albert.lsc.graph.automaton.State

METHODS

- *getFinal*
  `public static final DummyState` **getFinal( )**
  - **Usage**
    - Creates a new initial DummyState.
  - **Returns** - a DummyState d such that d.isDummyFinal() is true.

- *getInit*
  `public static final DummyState` **getInit( )**
  - **Usage**
    - Creates a new initial DummyState.
  - **Returns** - a DummyState d such that d.isDummyInit() is true.

- *isDummyFinal*
  `public boolean` **isDummyFinal( )**
  - **Usage**
    - Checks if this DummyState is a dummy final state.
  - **Returns** - true iff this State is (1) a dummy state and (2) final. Note that isDummyFinal() == ! isDummyInit().

- *isDummyInit*
  `public boolean` **isDummyInit( )**
  - **Usage**
    - Checks if this DummyState is a dummy initial state.
  - **Returns** - true iff this State is (1) a dummy state and (2) initial. Note that isDummyInit() == ! isDummyFinal().

- *toString*
  `public String` **toString( )**

1.1.4 CLASS State

A State in an Automaton. Every State knows which transitions start from itself.

DECLARATION

public class State
extends be.ac.fundp.info.albert.lsc.graph.SimpleVertex

CONSTRUCTORS

- *State*
  `public` **State(** `java.lang.Comparable` **id)**

METHODS

- *addTransition*
  `public void` **addTransition(**
  `be.ac.fundp.info.albert.lsc.graph.automaton.Transition t )`
- *cleanTransitionTable*
  `public void` **cleanTransitionTable(** `java.util.Collection c )`
- *compareTo*
  `public int` **compareTo(** `java.lang.Object o )`
- *equals*
  `public boolean` **equals(** `java.lang.Object o )`
- *getSuccessorOnLabel*
  `public Collection` **getSuccessorOnLabel(** `java.lang.Comparable` **label)**
- *getTransitionsLabels*
  `public Collection` **getTransitionsLabels(** **)**
- *isDummyFinal*
  `public boolean` **isDummyFinal(** **)**
  - **Usage**
    - checks if this State is a dummy final state.
  - **Returns** - true iff this State is a dummy final state.
- *isDummyInit*
  `public boolean` **isDummyInit(** **)**
  - **Usage**
    - checks if this State is a dummy initial state.
  - **Returns** - true iff this State is a dummy initial state.
- *numberOfTransitionsLabeled*
  `public int` **numberOfTransitionsLabeled(** `java.lang.Comparable` **label)**
- *numberOfTransitionsLabeled*
  `public int` **numberOfTransitionsLabeled(** `java.lang.Comparable` **label,**
  `java.util.Collection` **col)**
- *removeTransition*
  `public void` **removeTransition(**
  `be.ac.fundp.info.albert.lsc.graph.automaton.Transition t )`

1.1.5 CLASS Transition

an Object-labeled transition to use in an automaton. If the label is null, it corresponds to an epsilon-transition, in the automata-theoretic framework.

The labels used must be Comparable, in order to obtain efficient access to the transitions related to a state, in an automaton. The transitions are then ordered according to the lexicographical order on (State,Label). Two transitions (s1,l1),(s2,l2) are identical iff s1=s2 & l1=l2

DECLARATION

public class Transition
extends java.lang.Object
implements be.ac.fundp.info.albert.lsc.graph.Edge, java.lang.Comparable

CONSTRUCTORS

- *Transition*
  `public` **Transition(** `be.ac.fundp.info.albert.lsc.graph.automaton.State` **destination,** `java.lang.Comparable` **label)**
  - **Usage**
    - Creates a new Transition. Note that destination and label are mandatory information in a Transition.
  - **Parameters**
    - `destination` - the destination State of this Transition. May not be null.
    - `label` - the label of this Transition.

METHODS

- *compareTo*
  `public int` **compareTo(** `java.lang.Object` **o)**
  - **Usage**
    - Compares this Transition with the Transition o. o may not be null. The comparison is made according to the lexicographical order on (State,Label).
  - **Parameters**
    - `the` - Transition to compare with this Transition.
  - **Returns** - 0 iff o = this; -1 iff this <o; +1 iff this >0;

- *equals*
  `public boolean` **equals(** `java.lang.Object` **o)**
  - **Usage**
    - checks whether two transitions are equals.
  - **Parameters**
    - `o` - the Transition to test with this.
  - **Returns** - true iff the destination State and the label of this is equal to (using equals()) the destination State and the label of o
  - **Exceptions**
    - `ClassCastException` - if o is not a Transition.

- *getDestination*
  `public Vertex` **getDestination(** **)**
  - **Returns** - the Vertex (always a State) in which this Transition arrives.

- *getLabel*
  `public Comparable` **getLabel( )**
  - **Returns** - the label of this Transition.

- *hashCode*
  `public int` **hashCode( )**
  - **Returns** - a hash code for this transition. This hashcode is computed by taking the hash code of the String representation for this Transition.

- *isNull*
  `public boolean` **isNull( )**
  - **Usage**
    - tests if this Transition is a Null transition (i.e. an epsilon-transition, in the sense of the automata-theoretic framework).
  - **Returns** - true iff this is a null transition.

- *setDestination*
  `public void` **setDestination(** `be.ac.fundp.info.albert.lsc.graph.automaton.State s` **)**
  - **Usage**
    - Updates the destination State of this Transition
  - **Parameters**
    - `s` - the new destination State.

- *setLabel*
  `public void` **setLabel(** `java.lang.Comparable` **label)**
  - **Usage**
    - Updates the label of this Transition.
  - **Parameters**
    - `label` - the new value of the Transition's label.

- *toString*
  `public String` **toString( )**
  - **Usage**
    - Formats the Transition into a displayable form.
  - **Returns** - a human-readable description of this Transition.

Chapter 2

Package be.ac.fundp.info.albert.lsc.graph

Package Contents *Page*

Interfaces

Classes

2.1 Interfaces

2.1.1 Interface Edge

an Edge in a Graph. For the sake of simplicity, it only allows the retrieving of the Vertex in which this Edge arrives and not the Vertex from where it starts.

Declaration

public interface Edge

Methods

- *compareTo*
  `public int` **compareTo**`( java.lang.Object o )`
  - **Usage**
    - allows a comparison between two edges, in order to sort them.
  - **Parameters**
    - `o` - the object to which this object must be compared.
  - **Returns** - a negative, zero or positive integer if this object is, respectively less than, equals to or greater than `o`.
- *getDestination*
  `public Vertex` **getDestination**`( )`
  - **Returns** - the Vertex in which this Edge arrives.

2.1.2 Interface Vertex

This interface represents a Vertex in a graph. a Vertex must offer the ability to get the list of edges starting from it.
These edges are returned in a list, hence, they are ordered (by order of appearance in the graph : the first edge is the leftmost edge starting from the considered Vertex).

Declaration

public interface Vertex

Methods

- *compareTo*
  `public int` **compareTo**`( java.lang.Object o )`
  - **Usage**

* allows a comparison between two vertices, in order to sort them.
- **Parameters**
  * o - the object to which this object must be compared.
- **Returns** - a negative, zero or positive integer if this object is, respectively less than, equals to or greater than o.

- *getEdges*
  `public Vector` **getEdges**`( )`
  - **Returns** - a Vector of Edge containing all the transitions starting from this Vertex

2.2 Classes

2.2.1 Class Graph

This class represents a Graph. A graph is a set of Vertex, each Vertex "knowing" which are its successors in the graph. Every method provided by Graph restricts itself to the set of vertices contained in the graph. In particular, if the graph contains three vertices (v1,v2,v3) and, from v1, there is an edge leading to v2 and another vertex v4, the methods of Graph will ignore v4.

Declaration

public class Graph **extends** java.lang.Object

Constructors

- *Graph*
  `public` **Graph**`( )`
  - **Usage**
    * Creates a new, empty, Graph.

- *Graph*
  `public` **Graph**`( java.util.Collection c )`
  - **Usage**
    * Creates a Graph, with c has the initial vertices set.
  - **Parameters**
    * c - the initial set of Vertices.

Methods

- *addVertex*
  `public void` **addVertex**`( be.ac.fundp.info.albert.lsc.graph.Vertex v )`
  - **Usage**

* adds a vertex to this Graph. This Vertex must be of the same type as all the other vertices used in the graph.

– **Parameters**

* v - the vertex to add.

- *containsVertex*
`public boolean` **containsVertex(** `be.ac.fundp.info.albert.lsc.graph.Vertex v )`

– **Returns** - true iff vertices contains the vertex v.

- *getAcyclic*
`public Vector` **getAcyclic( )**

– **Usage**

* See the MSc Thesis [Bon2001], in the theoretical development of the translation algorithm, to get a full explanation of the fix-point algorithm to detect cycles in the transitive closure of a binary relation.

– **Returns** - acyclic, the Vector of vertex such that, if there is a vertex on which a cycle can be built, this vertex does not belong to acyclic.

- *getAllPaths*
`public Vector` **getAllPaths(** `be.ac.fundp.info.albert.lsc.graph.Vertex a, be.ac.fundp.info.albert.lsc.graph.Vertex b )`

– **Usage**

* Finds all possible paths between the Vertex a (belonging to vertices) and the Vertex b (belonging to vertices).

– **Parameters**

* a - the "origin" vertex
* b - the "destination" vertex

– **Returns** - a Vector containing all the possible and distinct paths in this Graph from a to b. Of course, these paths only use vertices from the set of vertices of this Graph.

- *getCycle*
`public Vector` **getCycle( )**

– **Returns** - a path that it a cycle. In other words, the last transition of this path leads to the state from where the first transition initiated. If there is no such path, returns null.

- *getLeftMostPath*
`public Vector` **getLeftMostPath(** `be.ac.fundp.info.albert.lsc.graph.Vertex a, be.ac.fundp.info.albert.lsc.graph.Vertex b )`

– **Usage**

* Compute the Left-Most path from a to b. A Left-Most path is a path that uses the first edge in every state (according to the order of edges as specified by getEdges()) to go from the initial to the final vertex. This path may not used any of the vertices contained in col.

– **Parameters**

  * `a` - the initial vertex (must belong to the set of vertices of the graph)
  * `b` - the final vertex (must belong to the set of vertices of the graph)
- **Returns** - a path (i.e. a Vector of Edges) leading from a to b, in a Left-Most way.

- *getLeftMostPath*
`public Vector` **getLeftMostPath**`( be.ac.fundp.info.albert.lsc.graph.Vertex a, be.ac.fundp.info.albert.lsc.graph.Vertex b, java.util.Collection col )`
  - **Usage**
    * Compute the Left-Most path from a to b, in this Graph. A Left-Most path is a path that uses the first edge in every state (according to the order of edges as specified by getEdges()) to go from the initial to the final vertex. This path may not use any of the vertices contained in col.
  - **Parameters**
    * `a` - the initial vertex (must belong to the set of vertices of the graph)
    * `b` - the final vertex (must belong to the set of vertices of the graph)
    * `col` - the path returned by this method does not use any of the vertices contained in col.
  - **Returns** - a path (i.e. a Vector of Edges) leading from a to b, in a Left-Most way.

- *getNextLeftMostPath*
`public Vector` **getNextLeftMostPath**`( java.util.Vector previousPath, be.ac.fundp.info.albert.lsc.graph.Vertex a, be.ac.fundp.info.albert.lsc.graph.Vertex b )`
  - **Usage**
    * Given a path from a Vertex to another Vertex, find the path that is directly greater than it in this graph, according to the lexicographical between paths. This lexicographical can be summarized as follows : (t1,...,tn) <(t'1,...,t'm) if t1 <t'm or if t1 = t'1 and (t2,...,tn) <(t'2,...,t'm). As a base, we consider that the empty path is smaller than every other non-empty path.

      Not implemented yet !!
  - **Parameters**
    * `previousPath` - the path from a to b that is directly smaller than the path to compute.
    * `a` - the initial Vertex of the path to find.
    * `b` - the final Vertex of the path to find.
  - **Returns** - the path from a to b that is directly greater than previousPath, according to the lexicographical order amongst paths in a Graph.

- *getPathBFS*
`public Vector` **getPathBFS**`( be.ac.fundp.info.albert.lsc.graph.Vertex from, be.ac.fundp.info.albert.lsc.graph.Vertex to )`
  - **Usage**
    * Find the shortest path from "from" to "to", using Breadth-First Search.

      Pre : "from" and "to" belong to vertices

      Post : everything is left unchanged. The path returned uses only the vertices of the graph.

- **Parameters**
  - `from` - the Vertex from which the path starts.
  - `to` - the Vertex in which the path arrives.
- **Returns** - the shortest path from "from" to "to". This path is a Vector composed of Edge. (e1,..,en) such that e1 starts from "from" and leads to a vertex "v1", belonging to the vertices of this graph. e2 starts from "v1" and leads to a vertex belonging to the vertices of this graph, called "v2". And so on, until en that leads to "to". If there is no path from "from" to "to", returns null.

- *getPathDFS*
  `public Vector` **getPathDFS(** `be.ac.fundp.info.albert.lsc.graph.Vertex` **from,** `be.ac.fundp.info.albert.lsc.graph.Vertex` **to)**
  - **Usage**
    - Finds a path from "from" to "to", using Depth-First Search.

      Pre : "from" and "to" belong to vertices

      Post : everything is left unchanged. And the path returned uses only the vertices of the graph.
  - **Parameters**
    - `from` - the Vertex from which the path starts.
    - `to` - the Vertex in which the path arrives.
  - **Returns** - a path from "from" to "to". This path is a Vector composed of Edge. (e1,..,en) such that e1 starts from "from" and leads to a vertex "v1", belonging to the vertices of this graph. e2 starts from "v1" and leads to a vertex belonging to the vertices of this graph, called "v2". And so on, until en that leads to "to". If there is no path from "from" to "to", returns null.

- *getPredecessorsInGraph*
  `public Collection` **getPredecessorsInGraph(**
  `be.ac.fundp.info.albert.lsc.graph.Vertex` **v)**
  - **Usage**
    - Gives the set of predecessors of the given Vertex v in the graph.
  - **Parameters**
    - `v` - the Vertex of which we want to compute the predecessors.
  - **Returns** - a set of Vertex, contained in the set of Vertex of this Graph, such that, for every Vertex belonging to it, there is an edge between v and this Vertex.

- *getPredecessorsInGraph*
  `public Collection` **getPredecessorsInGraph(**
  `be.ac.fundp.info.albert.lsc.graph.Vertex` **v,** `java.util.Collection` **ignore)**
  - **Usage**
    - Gives the set of predecessors of v in this Graph, that do not belong to the Collection of Vertex "ignore".

      This method is highly inefficient since it makes an heavy use of getSuccessorsInGraph(Vertex v) to compute the predecessors.

- **Parameters**
  - `v` - the Vertex of which we want to compute the predecessors.
  - `ignore` - the Collection of Vertex that may not include any of the predecessors returned by this method.
- **Returns** - a set of Vertex, contained in the set of Vertex of this Graph, such that, for every Vertex belonging to it, there is an edge between this Vertex and v, and this Vertex does not belong to ignore.

- *getSuccessorsInGraph*
  `public Collection` **getSuccessorsInGraph(**
  `be.ac.fundp.info.albert.lsc.graph.Vertex v )`
  - **Usage**
    - Gives the set of successors of the given Vertex v in the graph.
  - **Parameters**
    - `v` - the Vertex of which we want to compute the successors.
  - **Returns** - a set of Vertex, contained in the set of Vertex of this Graph, such that, for every Vertex belonging to it, there is an edge between v and this Vertex.

- *getSuccessorsInGraph*
  `public Collection` **getSuccessorsInGraph(**
  `be.ac.fundp.info.albert.lsc.graph.Vertex v, java.util.Collection ignore )`
  - **Usage**
    - Gives the set of successors of the given Vertex v in the graph that do not belong to ignore.
  - **Parameters**
    - `v` - the Vertex of which we want to compute the successors.
    - `the` - Collection of Vertex that may not include any of the successors returned by this method.
  - **Returns** - a set of Vertex, contained in the set of Vertex of this Graph, such that, for every Vertex belonging to it, there is an edge between v and this Vertex, and this Vertex does not belong to ignore.

- *getVertex*
  `public Vertex` **getVertex(** `be.ac.fundp.info.albert.lsc.graph.Vertex v )`
  - **Usage**
    - Get the Vertex in vertices, identical to v (according to equals()).
  - **Parameters**
    - `v` - the Vertex to retrieve in vertices.
  - **Returns** - the Vertex of vertices identical to v, if any. null, otherwise.

- *getVertices*
  `public Collection` **getVertices(** `)`
  - **Usage**
    - Gets all the vertices of this graph.
  - **Returns** - a Collection containing all the Vertices of this Graph.

- *removeVertex*
  `public void` **removeVertex(** `be.ac.fundp.info.albert.lsc.graph.Vertex v` **)**
  - **Usage**
    - removes the vertex v from the set of vertices of this graph.
  - **Parameters**
    - `v` - the Vertex to remove from this graph.

- *toString*
  `public String` **toString(** **)**
  - **Usage**
    - Format the graph for textual display.
  - **Returns** - a human-readable textual presentation of the graph.

2.2.2 Class SimpleVertex

SimpleVertex is a class representing a Vertex whose identifier is an Object. It is a generic Vertex. It is also an Edge. Hence, directed graphs can be represented using this class.

Declaration

public class SimpleVertex
extends java.lang.Object
implements Vertex, Edge, java.lang.Comparable

Constructors

- *SimpleVertex*
  `public` **SimpleVertex(** `java.lang.Comparable id` **)**
  - **Parameters**
    - `id` - the id of the new SimpleVertex

Methods

- *addEdge*
  `public void` **addEdge(** `be.ac.fundp.info.albert.lsc.graph.Edge v` **)**
  - **Usage**
    - Add a successor to this SimpleVertex.
  - **Parameters**
    - `v` - the Edge to add to this SimpleVertex.

- *compareTo*
  `public int` **compareTo(** `java.lang.Object o` **)**

- **Usage**
  - allows a comparison between two vertices, in order to sort them.
- **Parameters**
  - `o` - the object to which this object must be compared.
- **Returns** - a negative, zero or positive integer if this object is, respectively less than, equals to or greater than o.

- *equals*
  `public boolean` **equals**`( java.lang.Object` **vertex** `)`
  - **Usage**
    - Checks whether two SimpleVertex are equals.
  - **Returns** - true iff vertex.id = this.id
  - **Exceptions**
    - `ClassCastException` - if vertex is not a SimpleVertex.

- *getDestination*
  `public Vertex` **getDestination**`( )`
  - **Usage**
    - When called, this is considered as an Edge, leading to this SimpleVertex.
  - **Returns** - this

- *getEdges*
  `public Vector` **getEdges**`( )`
  - **Returns** - next, the Vector of successor SimpleVertex. Warning : this returns a reference to the original data and not a copy of it.

- *getId*
  `public Comparable` **getId**`( )`
  - **Returns** - a copy of this.id

- *hashCode*
  `public int` **hashCode**`( )`
  - **Returns** - the hash code for the identifier of this SimpleVertex

---

- *main*
  `public static final void` **main**`( java.lang.String [] argv )`

- *removeEdge*
  `public void` **removeEdge**`( be.ac.fundp.info.albert.lsc.graph.Edge` **v** `)`
  - **Usage**
    - Removes the SimpleVertex v from the list of successors of this SimpleVertex.
  - **Parameters**
    - `v` - the SimpleVertex to remove from the list of successors of this SimpleVertex.

- *setId*
  `public void` **setId**`( java.lang.Comparable` **newId** `)`

- **Usage**
  - Updates the id of this SimpleVertex
- **Parameters**
  - `newId` - the new Object identifying this SimpleVertex

- *toString*
  `public String` **toString( )**
  - **Usage**
    - Transforms this SimpleVertex into a displayable String.
  - **Returns** - the String form of the id.

Chapter 3

Package be.ac.fundp.info.albert.lsc.parser

Package Contents *Page*

3.1 Interfaces

3.1.1 Interface Parser

Defines the main interface for parsing a chart from a textual description.

Declaration

public interface Parser

Methods

- *getDescription*
  `public String` **getDescription**`( )`
  - **Returns** - this parser's features description, to display in the GUI
- *getName*
  `public String` **getName**`( )`
  - **Returns** - the name of this parser, to display in the GUI
- *parseChart*
  `public LscChart` **parseChart**`( java.io.File` **in** `)`
  - **Usage**
    - Builds a chart from the given input.
  - **Parameters**
    - `in` - the file in which the textual description of the chart will be read.
  - **Exceptions**
    - `java.io.IOException` - if an I/O error occurs when reading the chart.
- *toString*
  `public String` **toString**`( )`

3.2 Classes

3.2.1 Class EventParser

Parses a structured file into an EventMapping.

Declaration

public class EventParser
extends java.lang.Object

CONSTRUCTORS

- *EventParser*
  `public` **EventParser**`( )`
  - **Usage**
    - Creates new EventParser

METHODS

- *parse*
  `public EventMapping` **parse**`( java.io.File f )`
  - **Usage**
    - Parses f into an EventMapping.

      f must obey the following syntax:

      eventmapping ::= (lsc_event CRLF proposition CRLF)* END. CRLF

      lsc_event ::= char* proposition ::= char*
  - **Parameters**
    - `f` - the file containing the textual description of the mapping.
  - **Returns** - an EventMapping equivalent to the mapping described in f.
  - **Exceptions**
    - `be.ac.fundp.info.albert.lsc.parser.ParsingException` - if f is not compliant with the above syntax.
    - `java.io.IOException` - if an I/O Error occurs while reading f.

3.2.2 CLASS ParsingException

Exception thrown if the input of a parser is not compliant with the grammar fixed for the parser.

DECLARATION

public class ParsingException
extends java.lang.Exception

CONSTRUCTORS

- *ParsingException*
  `public` **ParsingException**`( )`
- *ParsingException*
  `public` **ParsingException**`( java.lang.Exception e )`
- *ParsingException*
  `public` **ParsingException**`( java.lang.String m )`

3.2.3 Class XmlBeParser

This parser takes as an input an LSC chart decsribed in the "belgian" XML format. It is not a parser for the "Weizmann" format of Rami.

Declaration

public class XmlBeParser
extends be.ac.fundp.info.albert.lsc.parser.XmlParser

Constructors

- *XmlBeParser*
  `public` **XmlBeParser**`( )`

Methods

- *getDescription*
  `public String` **getDescription**`( )`
- *getName*
  `public String` **getName**`( )`
- *main*
  `public static void` **main**`( java.lang.String [] argv )`
  - **Usage**
    * For debugging purpose only.
- *parseChart*
  `public LscChart` **parseChart**`( org.w3c.dom.Document` **doc** `)`

3.2.4 Class XmlParser

This class defines how an XML parser should work. It does all the parsing from a textual (XML) file to a DOM (in-memory) Document but leaves the transformation of the DOM into a chart to its specializations since this operation depends on the grammar chosen to describe the charts.

Declaration

public abstract class XmlParser
extends java.lang.Object
implements Parser

CONSTRUCTORS

- *XmlParser*
  `public` **XmlParser**`( )`

METHODS

- *getDescription*
  `public String` **getDescription**`( )`
- *getName*
  `public String` **getName**`( )`
- *parseChart*
  `public abstract LscChart` **parseChart**`( org.w3c.dom.Document` **doc** `)`
- *parseChart*
  `public LscChart` **parseChart**`( java.io.File` **in** `)`
  - **Usage**
    - Builds a chart from the given input.
  - **Parameters**
    - `in` - the input channel, from which the textual description of the chart will be read.
  - **Exceptions**
    - `java.io.IOException` - if an I/O error occurs when reading the chart.
- *toString*
  `public String` **toString**`( )`

Chapter 4

Package be.ac.fundp.info.albert.lsc.gui

Package Contents *Page*

Classes

4.1 Classes

4.1.1 Class **BrowseEvent**

This Event represents a click on a "Browse..." button.

Declaration

public class BrowseEvent
extends be.ac.fundp.info.albert.lsc.gui.GuiEvent

Constructors

- *BrowseEvent*
  `public` **BrowseEvent**`( java.lang.Object` **sender** `)`
  - **Usage**
    - Creates new BrowseEvent

4.1.2 Class **EventManager**

This is the conversation managers. It is also the translation sotware's main class. It receives all the events occurring in the GUI and manage them, by changing its internal state and acting on the GUI.

Declaration

public class EventManager
extends java.lang.Object

Constructors

- *EventManager*
  `public` **EventManager**`( )`
  - **Usage**
    - Creates new EventManager

Methods

- *getEvent*
  `public void` **getEvent**`( be.ac.fundp.info.albert.lsc.gui.BrowseEvent` **event** `)`
  - **Usage**
    - Reacts to a click on a "Browse..." button.

- **Parameters**
  - `event` - the event raised by the click on the "Browse..." button.

- *getEvent*
  `public void` **getEvent**`( be.ac.fundp.info.albert.lsc.gui.GuiEvent` **event** `)`
  - **Usage**
    - Reacts to a GUI event not managed by a more specialized method. Normally, this method should never be called.

- *getEvent*
  `public void` **getEvent**`( be.ac.fundp.info.albert.lsc.gui.NextEvent` **event** `)`
  - **Usage**
    - Reacts to a click on a "Next" button.
  - **Parameters**
    - `event` - the event raised by the click on the "Next" button.

- *getEvent*
  `public void` **getEvent**`( be.ac.fundp.info.albert.lsc.gui.PreviousEvent` **event** `)`
  - **Usage**
    - Reacts to a click on a "Previous" button.
  - **Parameters**
    - `event` - the event raised by the click on the "previous" button.

- *main*
  `public static void` **main**`( java.lang.String [] args )`
  - **Usage**
    - The translation software main method. In order to launch the application, type "java be.ac.fundp.info.albert.lsc.gui.EventManager" in a prompt line. No command line arguments are needed nor used by this method.

4.1.3 Class GuiEvent

A generic Event in the interface. When an event occurs in the graphical part of the GUI (the presentation), the component sends a specialization of this class to the EventManager of the application. It will then react properly and update the GUI.

Declaration

public class GuiEvent
extends java.lang.Object

CONSTRUCTORS

- *GuiEvent*
  `public` **GuiEvent**`( java.lang.Object` **sender** `)`
  - **Usage**
    - Creates new GuiEvent
- *GuiEvent*
  `public` **GuiEvent**`( java.lang.Object` **sender**`, java.lang.Object` **param** `)`

METHODS

- *getParam*
  `public Object` **getParam**`( )`
- *getSender*
  `public Object` **getSender**`( )`

4.1.4 CLASS NextEvent

This Event represents a click on a "Next" button.

DECLARATION

public class NextEvent
extends be.ac.fundp.info.albert.lsc.gui.GuiEvent

CONSTRUCTORS

- *NextEvent*
  `public` **NextEvent**`( java.lang.Object` **sender** `)`
  - **Usage**
    - Creates new NextEvent

4.1.5 CLASS PreviousEvent

This Event represents a click on a "Previous" button.

DECLARATION

public class PreviousEvent
extends be.ac.fundp.info.albert.lsc.gui.GuiEvent

CONSTRUCTORS

- *PreviousEvent*
  `public` **PreviousEvent**`( java.lang.Object` **sender** `)`
  - **Usage**
    * Creates new PreviousEvent

4.1.6 CLASS SelectFile

This frame is a wizard allowing the user to select a file on disk, and possibly clicking on next or previous.

DECLARATION

public class SelectFile **extends** javax.swing.JFrame

SERIALIZABLE FIELDS

- private EventManager manager
  –
- private JTextArea explanationText
  –
- private JPanel jPanel1
  –
- private JButton browseButton
  –
- private JTextField fileText
  –
- private JButton previousButton
  –
- private JButton nextButton
  –

CONSTRUCTORS

- *SelectFile*
  `public` **SelectFile**`( be.ac.fundp.info.albert.lsc.gui.EventManager` **manager**`, java.lang.String` **title**`, java.lang.String` **explanation**`, boolean` **previousEnabled**`, boolean` **nextEnabled** `)`

METHODS

- *getSelection*
  `public String` **getSelection**`( )`
- *setSelection*
  `public void` **setSelection**`( java.lang.String s )`

4.1.7 CLASS SelectMode

This frame allows the user to choose a mode from a combobox. A mode can be an input or output formalism.

DECLARATION

public class SelectMode
extends javax.swing.JFrame

SERIALIZABLE FIELDS

- private EventManager manager
  –
- private JTextArea explanationText
  –
- private JButton previousButton
  –
- private JButton nextButton
  –
- private JPanel jPanel1
  –
- private JComboBox choicesCombo
  –

CONSTRUCTORS

- *SelectMode*
  `public` **SelectMode**`( be.ac.fundp.info.albert.lsc.gui.EventManager` **manager**`, java.lang.String` **title**`, java.lang.String` **explanation**`, java.util.Vector` **choices**`, boolean` **previousEnabled**`, boolean` **nextEnabled** `)`

METHODS.

- *getSelection*
  `public Object` **getSelection( )**

Chapter 5

Package be.ac.fundp.info.albert.lsc.syntax

Package Contents *Page*

Classes

5.1 Classes

5.1.1 Class **BadLscException**

This exception is thrown whenever an LSC violates a method's precondition. In particular, it is thrown if the LSC is not well-formed.

Declaration

public class BadLscException
extends java.lang.Exception

Constructors

- *BadLscException*
  `public` **BadLscException**`( )`
- *BadLscException*
  `public` **BadLscException**`( java.lang.String` **msg** `)`

5.1.2 Class **LscBasicChart**

This is a basic chart (i.e. whether a main chart or a prechart). It is composed of instance lines and of messages linking locations together.

Declaration

public class LscBasicChart
extends be.ac.fundp.info.albert.lsc.syntax.LscObject

Constructors

- *LscBasicChart*
  `public` **LscBasicChart**`( java.lang.String` **name** `)`
  - **Usage**
    - Creates a new BasicChart identified by name.
  - **Parameters**
    - name - the name of the basic chart to create.

METHODS

- *addInstance*
  `public void` **addInstance(** `be.ac.fundp.info.albert.lsc.syntax.LscInstance` **inst)**
  - **Usage**
    - Adds an instance line to this chart (inst != null).
  - **Parameters**
    - `inst` - the Instance line to add to this chart.

- *addMessage*
  `public void` **addMessage(** `be.ac.fundp.info.albert.lsc.syntax.LscMessage` **msg)**
  - **Usage**
    - Adds a message to this chart (msg != null).
  - **Parameters**
    - `msg` - the Message add to this chart.

- *getInstances*
  `public LscObjectSequence` **getInstances( )**
  - **Returns** - the instances of this chart.

- *getMessages*
  `public LscObjectSequence` **getMessages( )**
  - **Returns** - the messages of this chart

- *getReceiverBySender*
  `public LscLocation` **getReceiverBySender(** `be.ac.fundp.info.albert.lsc.syntax.LscLocation` **sender)**
  - **Usage**
    - Finds the location at which the message sent in location sender is received.
  - **Returns** - the location at which the message sent in location sender is received.

- *getSenderByReceiver*
  `public LscLocation` **getSenderByReceiver(** `be.ac.fundp.info.albert.lsc.syntax.LscLocation` **receiver)**
  - **Usage**
    - Finds the location at which the message received in location sender is sent.
  - **Returns** - the location at which the message received in location sender is sent.

5.1.3 CLASS LscChart

An activated chart. An activated chart is a triple (BasicChart,BasicChart,Set of restricted Events). It has a temperature determining its mode (universal,existential) and it can also describe a no-story.

DECLARATION

public class LscChart
extends be.ac.fundp.info.albert.lsc.syntax.LscModalObject

FIELDS

- public static final boolean EXIST
  - Constant to specify the mode of the chart (existential vs universal)
- public static final boolean UNIV
  - Constant to specify the mode of the chart (existential vs universal)
- public static final boolean NO_STORY
  - Constant to specify the type of chart (scenario vs anti-scenario)
- public static final boolean YES_STORY
  - Constant to specify the type of chart (scenario vs anti-scenario)

CONSTRUCTORS

- *LscChart*
  `public` **LscChart**`( java.lang.String` **name**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscBasicChart` **prechart**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscBasicChart` **mainchart**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscObjectSequence` **restr**`, boolean`
  temperature`, boolean` **yesStory** `)`
  - **Usage**
    - Constructs a new activated chart, identified by name.
  - **Parameters**
    - `name` - the identifier of the new chart.
    - `prechart` - the prechart of the new chart.
    - `mainchart` - the main chart of the new chart.
    - `restr` - a sequence containing all the events restricted in the chart.
    - `temperature` - in {EXIST,UNIV}, the temperature of the chart (determines its mode).
    - `yesStory` - in (YES_STORY,NO_STORY), the type of the chart.

METHODS

- *createExistentialChart*
  `public static LscChart` **createExistentialChart**`( java.lang.String` **name**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscBasicChart` **prechart**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscBasicChart` **mainchart**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscObjectSequence` **restrictedEvents** `)`

- **Usage**
  - Constructs a new existential chart, identified by name.
- **Parameters**
  - `prechart` - the prechart of the new chart.
  - `mainchart` - the main chart of the new chart.
  - `restr` - a sequence containing all the events restricted in the chart.

- *createNoStoryChart*

```
public static LscChart createNoStoryChart( java.lang.String name,
be.ac.fundp.info.albert.lsc.syntax.LscBasicChart prechart,
be.ac.fundp.info.albert.lsc.syntax.LscBasicChart mainchart,
be.ac.fundp.info.albert.lsc.syntax.LscObjectSequence restrictedEvents )
```

  - **Usage**
    - Constructs a new existential chart, describing a no-story, identified by name.
  - **Parameters**
    - `prechart` - the prechart of the new chart.
    - `mainchart` - the main chart of the new chart.
    - `restr` - a sequence containing all the events restricted in the chart.

- *createUniversalChart*

```
public static LscChart createUniversalChart( java.lang.String name,
be.ac.fundp.info.albert.lsc.syntax.LscBasicChart prechart,
be.ac.fundp.info.albert.lsc.syntax.LscBasicChart mainchart,
be.ac.fundp.info.albert.lsc.syntax.LscObjectSequence restrictedEvents )
```

  - **Usage**
    - Constructs a new universal chart, identified by name.
  - **Parameters**
    - `prechart` - the prechart of the new chart.
    - `mainchart` - the main chart of the new chart.
    - `restr` - a sequence containing all the events restricted in the chart.

- *getMainChart*

```
public LscBasicChart getMainChart( )
```

  - **Returns** - the main chart of this chart.

- *getPrechart*

```
public LscBasicChart getPrechart( )
```

  - **Returns** - the prechart of this chart.

- *getRestrictedEvents*

```
public LscObjectSequence getRestrictedEvents( )
```

  - **Returns** - the events restricted in this chart.

- *isExistential*

```
public boolean isExistential( )
```

  - **Returns** - true iff this chart is Existential (describes a safety property).

- *isNoStory*
  `public boolean` **isNoStory**`( )`
  - **Returns** - true iff this chart is a No-story (describes a safety negative property).

- *isUniversal*
  `public boolean` **isUniversal**`( )`
  - **Returns** - true iff this chart is Universal (describes a property of liveness).

- *toString*
  `public String` **toString**`( )`

5.1.4 Class LscCut

A cut in a chart is a set of Locations containing at least one Location per Instance.

Declaration

public class LscCut
extends java.lang.Object
implements java.lang.Comparable

Constructors

- *LscCut*
  `public` **LscCut**`( be.ac.fundp.info.albert.lsc.syntax.LscBasicChart` **chart** `)`
  - **Usage**
    - Creates the initial cut of a given chart. The initial cut is the cut containing only the initial location of every instance of the chart.
  - **Parameters**
    - `chart` - the chart of which we want to build the initial cut.

Methods

- *buildSuccessor*
  `public LscCut` **buildSuccessor**`( be.ac.fundp.info.albert.lsc.syntax.LscLocation` **loc** `)`
  - **Usage**
    - Builds the loc-successor of this cut. The loc-successor of a cut c is the cut that differs from c only by the addition of the location loc.
  - **Parameters**
    - `loc` - the location to add at this cut to create its successor.

- *compareTo*
  `public int` **compareTo**`( java.lang.Object o )`

- *containsLocation*
  `public boolean` **containsLocation**`(`
  `be.ac.fundp.info.albert.lsc.syntax.LscLocation` **loc** `)`
  - **Usage**
    - Checks whether this cut contains the location loc. The method "equals" in LscLocation in used to perform the test.
  - **Parameters**
    - `loc` - the location to search for in this cut.
  - **Returns** - true iff this cut contains loc.

- *equals*
  `public boolean` **equals**`( java.lang.Object o )`

- *getLocations*
  `public Collection` **getLocations**`( )`
  - **Returns** - the set of locations of this cut.

- *hashCode*
  `public int` **hashCode**`( )`

- *isFinal*
  `public boolean` **isFinal**`( )`
  - **Usage**
    - Checks whether or not this cut is a final cut. By definition, a cut is final if the "biggest" (here, by assumption, the most lately added) locations are all cold, in every instance.
  - **Returns** - true iff this cut is final.

- *toString*
  `public String` **toString**`( )`

5.1.5 Class LscEvent

An Event in a chart. In every Location of a chart, there is at most one event occurring. An Event is the receiving or the sending of a message. Every Event is related to an Instance.

Declaration

public abstract class LscEvent **extends** be.ac.fundp.info.albert.lsc.syntax.LscObject

METHODS

- *isReceive*
  `public boolean` **isReceive( )**
  - **Returns** - true iff this Event corresponds to the receiving of a message. Note that, by definition, isReceive() == !isSend().

- *isSend*
  `public abstract boolean` **isSend( )**
  - **Returns** - true iff this Event corresponds to the sending of a message.

- *setOwner*
  `public void` **setOwner(** `java.lang.String` **instanceOwner)**
  - **Usage**
    - Changes the owner of this Event.
  - **Parameters**
    - `instanceOwner` - the Instance becoming the new owner of this Event.

- *toString*
  `public String` **toString( )**

5.1.6 CLASS LscInstance

An Instance line in a chart. An instance line is identified by its instance (or thread, or process) name. It consists of an ordered sequence (according to the so-called visual order) of unordered locations. Unordered locations are gathered into what we call coregions.

DECLARATION

public class LscInstance
extends be.ac.fundp.info.albert.lsc.syntax.LscObject

CONSTRUCTORS

- *LscInstance*
  `public` **LscInstance(** `java.lang.String` **name)**
  - **Usage**
    - Creates a new LscInstance, with one Location, being initial and final.
  - **Parameters**
    - `name` - the name (identifier) of this Instance.

- *LscInstance*
  `public` **LscInstance**`( java.lang.String` **name**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscEvent` **evInit**`, boolean` **temperature**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscEvent` **evFinal** `)`
  - **Usage**
    * Creates a new LscInstance, with an initial and a final location.
  - **Parameters**
    * `name` - the name (identifier) of this Instance.
    * `evInit` - the Event related to the initial Location of this Instance (not null).
    * `temperature` - the temperature of the initial Location of this Instance.
    * `evFinal` - the Event related to the final Location of this Instance (not null).

METHODS

- *addCoregion*
  `public void` **addCoregion**`( be.ac.fundp.info.albert.lsc.syntax.LscLocation` **loc** `)`
  - **Usage**
    * Adds a new Coregion to the end of this Instance line, containing only one Location.
  - **Parameters**
    * `loc` - the Location that the new coregion must contain.

- *addCoregion*
  `public void` **addCoregion**`(`
  `be.ac.fundp.info.albert.lsc.syntax.LscObjectSequence` **coregion** `)`
  - **Usage**
    * Adds the given coregion to the end of this instance line
  - **Parameters**
    * `coregion` - the coregion to add at the end of this instance line.

- *addLocation*
  `public void` **addLocation**`( be.ac.fundp.info.albert.lsc.syntax.LscLocation`
  loc`, be.ac.fundp.info.albert.lsc.syntax.LscObjectSequence` **coRegion** `)`
  - **Usage**
    * Adds a given Location to the coregion coRegion.
  - **Parameters**
    * `loc` - the Location to add to the given coregion.
    * `coRegion` - the coregion that will contain loc.

- *createNewLocation*
  `public LscLocation` **createNewLocation**`( boolean` **temperature**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscEvent` **ev** `)`

- *getCoregions*
  `public LscObjectSequence` **getCoregions**`( )`

- **Returns** - the coregions (LscObjectSequence of LscObjectSequence of Location) of this Instance Line, ordered according to their visual order.

- *getInitialLocation*
  `public LscLocation` **getInitialLocation**`( )`
  - **Usage**
    - Finds the initial Location of this Instance Line. By the specification of the LSCs language, we require every instance line to have one and only one, not necessarily related to an event, initial location.
  - **Returns** - the initial Location of this Instance Line.

- *getMaxLocations*
  `public LscObjectSequence` **getMaxLocations**`( )`
  - **Returns** - the set of Maximal Locations of this Instance. These Locations must all be cold, as required in the specification of the LSCs language.

5.1.7 Class **LscLocation**

Declaration

public class LscLocation
extends be.ac.fundp.info.albert.lsc.syntax.LscModalObject

Constructors

- *LscLocation*
  `public` **LscLocation**`( boolean` **temperature**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscInstance` **owner**`, java.lang.String` **id**
  `)`
  - **Usage**
    - Creates a new initial Location.
  - **Parameters**
    - `temperature` - the temperature of this Location (see LscModalObject).
    - `owner` - the Instance owning this Location.
    - `id` - a local identifier for the new Location. A local identifier is a String such that there is no other Location belonging to the same Instance and having the same id.

- *LscLocation*
  `public` **LscLocation**`( boolean` **temperature**`,`
  `be.ac.fundp.info.albert.lsc.syntax.LscInstance` **ownerInst**`, java.lang.String`
  id`, be.ac.fundp.info.albert.lsc.syntax.LscEvent` **ev** `)`
  - **Usage**
    - Creates a new Location.
  - **Parameters**

* `temperature` - the temperature of this Location (see LscModalObject).
* `owner` - the Instance owning this Location.
* `id` - a local identifier for the new Location. A local identifier is a String such that there is no other Location belonging to the same Instance and having the same id.
* `ev` - the Event related to this Location. If null, throws BadLscException.

METHODS

- *getEvent*
  `public LscEvent` **`getEvent( )`**
- *getLocalId*
  `public String` **`getLocalId( )`**
- *getOwnerId*
  `public String` **`getOwnerId( )`**
- *isInitial*
  `public boolean` **`isInitial( )`**
- *setEvent*
  `public void` **`setEvent(`** `be.ac.fundp.info.albert.lsc.syntax.LscEvent` **`ev )`**
- *setLocalId*
  `public void` **`setLocalId(`** `java.lang.String` **`id )`**
- *setOwnerId*
  `public void` **`setOwnerId(`** `java.lang.String` **`id )`**

5.1.8 CLASS LscMessage

A message, in a chart representation. A message links two locations.

DECLARATION

public class LscMessage
extends be.ac.fundp.info.albert.lsc.syntax.LscModalObject

FIELDS

- public static final boolean SYNCH
  - Constant for specifying that a message is synchronous
- public static final boolean ASYNCH
  - Constant for specifying that a message is asynchronous

CONSTRUCTORS

- *LscMessage*
  `public` **LscMessage**`( java.lang.String` **msgName**`, be.ac.fundp.info.albert.lsc.syntax.LscLocation` **sender**`, be.ac.fundp.info.albert.lsc.syntax.LscLocation` **receiver**`, boolean` **typeComm** `)`
  - **Usage**
    - Creates a new instance of the message msgName, linking the location sender to the location receiver. The communication type of this message is typeComm
  - **Parameters**
    - `msgName` - the name of the class of messages the new message should belong to.
    - `sender` - the location at which this message is sent.
    - `receiver` - the location ar which this message is received.
    - `typeComm` - in {SYNCH,ASYNCH}. Determines if this message is synchronous or asynchronous.

METHODS

- *getReceiver*
  `public LscLocation` **getReceiver**`( )`
  - **Returns** - the location at which this message is received.

- *getSender*
  `public LscLocation` **getSender**`( )`
  - **Returns** - the location at which this message is sent.

- *isAsynch*
  `public boolean` **isAsynch**`( )`
  - **Returns** - true iff this message is asynchronous

- *isSynch*
  `public boolean` **isSynch**`( )`
  - **Returns** - true iff this message is synchronous

5.1.9 CLASS LscModalObject

An LSC object having the modal "temperature" property. It can be either cold (describing a provisional behavior) or hot (describing a mandatory behavior).

DECLARATION

public class LscModalObject
extends be.ac.fundp.info.albert.lsc.syntax.LscObject

FIELDS

- public static final boolean HOT
  - A constant representing the temperature of a hot Modal Object.
- public static final boolean COLD
  - A constant representing the temperature of a hot Modal Object.

CONSTRUCTORS

- *LscModalObject*
  `public` **LscModalObject**`( java.lang.String` **name**`, boolean` **temp** `)`
  - **Usage**
    - Creates a new Modal Object, called "name", having the temperature temp.
  - **Parameters**
    - `name` - the name (identifier) of this object.
    - `temp` - (in {HOT,COLD}), the temperature of this object.

METHODS

- *isCold*
  `public boolean` **isCold**`( )`
  - **Returns** - true iff this object is cold. Note that isCold() == !isHot().

- *isHot*
  `public boolean` **isHot**`( )`
  - **Returns** - true iff this object is hot. Note that isHot() == ! isCold().

- *makeCold*
  `public void` **makeCold**`( )`
  - **Usage**
    - Makes a cold object with this object. In other words, after makeCold(), isCold() is true.

- *makeHot*
  `public void` **makeHot**`( )`
  - **Usage**
    - Makes a hot object with this object. In other words, after makeHot(), isHot() is true.

- *toString*
  `public String` **toString**`( )`
  - **Returns** - a human-readable presentation of this object.

5.1.10 Class **LscObject**

The ancestor of every object used in the internal description of a chart. It defines the most basic properties that all the Syntax objects should enjoy.

Declaration

public class LscObject
extends java.lang.Object
implements java.lang.Comparable

Constructors

- *LscObject*
  `public` **LscObject**`( java.lang.String` **name** `)`
  - **Usage**
    - Creates a new object identified by name
  - **Parameters**
    - `name` - the identifier of the new object

Methods

- *compareTo*
  `public int` **compareTo**`( java.lang.Object` **o** `)`
  - **Usage**
    - Tests if this object is smaller than o. By definition (and if the specializations of this class do not decide to do it in a better way), an object a is smaller than an object b iff a's identifier is smaller than b's identifier.
  - **Parameters**
    - `o` - the object to which this object should be compared.
  - **Returns** - an integer that is less than, equal to or greater than zero if this object is, resp., smaller, equal to, or greater than o.

- *equals*
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Usage**
    - Checks whether or not two objects are equals. By definition, two objects are equals iff their identifier are the same.
  - **Parameters**
    - `o` - the object this object should be compared to.
  - **Returns** - true iff this object is equals to o.

- *getName*
  `public String` **getName( )**
  - **Returns** - this object's identifier.

- *hashCode*
  `public int` **hashCode( )**
  - **Returns** - a Hash code for this object, computed from its identifier.

- *setName*
  `public void` **setName(** `java.lang.String` **name)**
  - **Usage**
    - Modified this object's name to be equal to name.
  - **Parameters**
    - `name` - this object's new name.

- *toString*
  `public String` **toString( )**
  - **Returns** - a String representation (preferably in a human-readable notation) of this object.

5.1.11 Class LscObjectSequence

A sequence of LscObject.

Declaration

public class LscObjectSequence **extends** java.util.Vector

Constructors

- *LscObjectSequence*
  `public` **LscObjectSequence( )**
  - **Usage**
    - Creates a new empty Sequence

- *LscObjectSequence*
  `public` **LscObjectSequence(** `java.util.Collection` **col)**
  - **Usage**
    - Creates a new Sequence containing the Collection col.
  - **Parameters**
    - `col` - the population of the new Sequence.

- *LscObjectSequence*
  `public` **LscObjectSequence**`( be.ac.fundp.info.albert.lsc.syntax.LscObject` **o** `)`
  - **Usage**
    - Creates a new Sequence containing o
  - **Parameters**
    - `o` - the element to add to the new Sequence.

METHODS

- *add*
  `public boolean` **add**`( be.ac.fundp.info.albert.lsc.syntax.LscObject` **o** `)`
  - **Usage**
    - Adds an LscObject to this Sequence.
  - **Parameters**
    - `o` - the object to add to this sequence.

- *getObjectByName*
  `public LscObject` **getObjectByName**`( java.lang.String` **name** `)`
  - **Usage**
    - Retrieves the first LscObject called "name" in this Sequence.
  - **Parameters**
    - `name` - the name (identifier) of the LscObject to retrieve.
  - **Returns** - the first occurrence of an LscObject called name in this Sequence, if any. null, otherwise.
  - **Exceptions**
    - `ClassCastException` - if this Sequence contains an Object which is not an LscObject.

5.1.12 CLASS LscReceiveEvent

A receive Event. It is an Event in which a message is received by the Instance owning this Event.

DECLARATION

public class LscReceiveEvent
extends be.ac.fundp.info.albert.lsc.syntax.LscEvent

CONSTRUCTORS

- *LscReceiveEvent*
  `public` **LscReceiveEvent**`( java.lang.String` **name**`, java.lang.String` **receiver** `)`

METHODS

- *getReceiver*
  `public String` **getReceiver**`( )`
  - **Returns** - the identifier of the Instance receiving the message of this Event.

- *isSend*
  `public boolean` **isSend**`( )`

5.1.13 CLASS LscSendEvent

A send Event. It is an Event, owned by an Instance, in which a message is sent.

DECLARATION

public class LscSendEvent
extends be.ac.fundp.info.albert.lsc.syntax.LscEvent

CONSTRUCTORS

- *LscSendEvent*
  `public` **LscSendEvent**`( java.lang.String` **name**`, java.lang.String` **sender** `)`

METHODS

- *getSender*
  `public String` **getSender**`( )`
  - **Usage**
    * The Instance sending this Event.

- *isSend*
  `public boolean` **isSend**`( )`

5.1.14 CLASS LscSpecification

DECLARATION

public class LscSpecification
extends be.ac.fundp.info.albert.lsc.syntax.LscObject

CONSTRUCTORS

- *LscSpecification*
  `public` **LscSpecification**`( java.lang.String` **name** `)`

METHODS

- *addChart*
  `public void` **addChart**`( be.ac.fundp.info.albert.lsc.syntax.LscChart` **chart** `)`
- *main*
  `public static final void` **main**`( )`
- *removeChart*
  `public void` **removeChart**`( be.ac.fundp.info.albert.lsc.syntax.LscChart` **chart** `)`

Chapter 6

Package be.ac.fundp.info.albert.lsc.verification

Package Contents *Page*

6.1 Interfaces

6.1.1 INTERFACE EventMapping

This defines how an event (in the terms of the inter-object specification) is mapped onto a Proposition (in the terms of the model-checker input language). This interface is thus used for generating the temporal logic formula (or any other proof file for a model-checker). In other words, this interface carries out the work for generation (from scenario to model-checker).

In order to be able to save (load) a mapping to (from) a file, a mapping must be serializable.

DECLARATION

public interface EventMapping
implements java.io.Serializable

METHODS

- *eventToProp*
  `public String` **eventToProp**`( be.ac.fundp.info.albert.lsc.syntax.LscEvent event )`
  - **Usage**
    - Maps an event (in the inter-object sense) to the corresponding proposition, (in the intra-object sense).
  - **Parameters**
    - `event` - the (inter-object) Event of which we want to retrieve the corresponding proposition.
  - **Returns** - the (intra-object) proposition to which event corresponds.

6.1.2 INTERFACE ProofGenerator

A class to generate a proof-file adapted to a special kind of model-checker must implement this interface. It will provide three methods to generate three different kinds of proof-file, for universal, existential or no-story scenarios.

DECLARATION

public interface ProofGenerator

METHODS

- *generateExistentialProof*
  `public void` **generateExistentialProof(**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart,**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart,**
  `java.util.Collection` **restricted,**
  `be.ac.fundp.info.albert.lsc.verification.EventMapping` **map,**
  `java.io.OutputStream` **out)**
  - **Usage**
    * Writes to an outputStream the proof file of an existential scenario. In the terms of [Bon2001], prechart is the automaton A$_{cuts(prech(m))}$, and appended is the automaton A$_append(m)$, i.e. its language is the trace set of the main chart appended to the trace set of the prechart (L(prechart).L(mainchart)). The proof file generated uses the proposition mapping in map and is written on outputStream.
  - **Parameters**
    * `prechart` - the TraceSet of the prechart.
    * `mainChart` - the TraceSet of the main chart.
    * `restricted` - the set of restricted events of the chart.
    * `map` - a mapping from LscEvent to Propositions (i.e. String).
    * `out` - the outputStream on which the proof file must be written.

- *generateNoStoryProof*
  `public void` **generateNoStoryProof(**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart,**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart,**
  `java.util.Collection` **restricted,**
  `be.ac.fundp.info.albert.lsc.verification.EventMapping` **map,**
  `java.io.OutputStream` **out)**
  - **Usage**
    * Writes to an outputStream the proof file of a no-story scenario (anti-scenario). In the terms of [Bon2001], prechart is the automaton A$_{cuts(prech(m))}$, and appended is the automaton A$_append(m)$, i.e. its language is the trace set of the main chart appended to the trace set of the prechart (L(prechart).L(mainchart)). The proof file generated uses the proposition mapping in map and is written on outputStream.
  - **Parameters**
    * `prechart` - the TraceSet of the prechart.
    * `mainChart` - the TraceSet of the main chart.
    * `restricted` - the set of restricted events of the chart.
    * `map` - a mapping from LscEvent to Propositions (i.e. String).
    * `out` - the outputStream on which the proof file must be written.

- *generateUniversalProof*
  `public void` **generateUniversalProof(**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart,**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart,**
  `java.util.Collection` **restricted,**

`be.ac.fundp.info.albert.lsc.verification.EventMapping` **map,**
`java.io.OutputStream` **out**)

- **Usage**
  - Writes to an outputStream the proof file of a universal scenario. In the terms of [Bon2001], prechart is the automaton A$_{cuts(prech(m))}$, and appended is the automaton A$_append(m)$, i.e. its language is the trace set of the main chart appended to the trace set of the prechart (L(prechart).L(mainchart)). The proof file generated uses the proposition mapping in map and is written on outputStream.
- **Parameters**
  - `prechart` - the TraceSet of the prechart.
  - `mainChart` - the TraceSet of the main chart.
  - `restricted` - the set of restricted events of the chart.
  - `map` - a mapping from LscEvent to Propositions (i.e. String).
  - `out` - the outputStream on which the proof file must be written.

- *toString*
  `public String` **toString**`( )`

6.1.3 INTERFACE PropMapping

This defines how a Proposition (in the terms of the inter-object specification) is mapped onto a Proposition (in the terms of the model-checker input language). This interface is thus used for transforming an output from the model-checker into a human-readable scenario (LscObjectSequence of LscEvent). In other words, this interface carries out the work for feedback (from model-checker to scenario).

In order to be able to save (load) a mapping to (from) a file, a mapping must be serializable.

DECLARATION

public interface PropMapping **implements** java.io.Serializable

METHODS

- *parseInput*
  `public LscObjectSequence` **parseInput**`( java.io.InputStream` **inputStream**)
  - **Usage**
    - Parses an input (a result from a model-checker, for example), and transforms it into a sequence of LscEvent. This sequence of Event could then be used to construct an existential scenario (if the model-checker sent back a counter-example).
  - **Parameters**

  * `inputStream` - the result of the model-checker to transform into a sequence of LscEvent.
- **Returns** - an LscObjectSequence containing LscEvent, representing the input, in the terms of the inter-object specification.
- **See Also**
  * `be.ac.fundp.info.albert.lsc.syntax.LscEvent` (in 5.1.5, page 168)
  * `be.ac.fundp.info.albert.lsc.verification.EventMapping` (in 6.1.1, page 182)

6.2 Classes

6.2.1 Class CausalOrder

The causal order ($<_m$) is represented by a directed graph in which the set of vertices is the set of locations of the chart (dom(m)) and there is a path from a to b iff $b <_m a$.

Declaration

public class CausalOrder
extends be.ac.fundp.info.albert.lsc.graph.Graph

Constructors

- *CausalOrder*
  `public` **CausalOrder(** `be.ac.fundp.info.albert.lsc.syntax.LscBasicChart` **chart**)

Methods

- *getMinLocations*
  `public Vector` **getMinLocations(** `java.util.Collection` **ignore**)
  - **Usage**
    * Gets the smallest Locations in dom(m) - ignore. In the terms of [Bon2001], computes $L_{“bot}$.
  - **Parameters**
    * `ignore` - a set of Location not to take into account when computing the smallest events.
  - **Returns** - a Vector of LscLocations such that, for every Location l in the chart, either l belongs to ignore or l is greater than every Location in this Vector.

6.2.2 Class CTLStarGenerator

This class is a generator for CTL* formulae. The formulae generated are pure-text formulae and do not aim at be used by any model-checker as is.

DECLARATION

public class CTLStarGenerator
extends java.lang.Object
implements ProofGenerator

CONSTRUCTORS

- *CTLStarGenerator*
  `public` **CTLStarGenerator**( )

METHODS

- *generateExistentialProof*
  `public void` **generateExistentialProof**(
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart**,
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart**,
  `java.util.Collection` **restricted**,
  `be.ac.fundp.info.albert.lsc.verification.EventMapping` **map**,
  `java.io.OutputStream` **out**)
- *generateNoStoryProof*
  `public void` **generateNoStoryProof**(
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart**,
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart**,
  `java.util.Collection` **restricted**,
  `be.ac.fundp.info.albert.lsc.verification.EventMapping` **map**,
  `java.io.OutputStream` **out**)
- *generateUniversalProof*
  `public void` **generateUniversalProof**(
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart**,
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart**,
  `java.util.Collection` **restricted**,
  `be.ac.fundp.info.albert.lsc.verification.EventMapping` **map**,
  `java.io.OutputStream` **out**)
- *toString*
  `public String` **toString**( )

6.2.3 CLASS IdentityEventMapping

For every event, this maps it to its string format.

DECLARATION

public class IdentityEventMapping
extends java.lang.Object
implements EventMapping

CONSTRUCTORS

- *IdentityEventMapping*
  `public` **IdentityEventMapping**`( )`

METHODS

- *eventToProp*
  `public String` **eventToProp**`( be.ac.fundp.info.albert.lsc.syntax.LscEvent` **event** `)`
  - **Usage**
    - Returns, for the given non-null event, its String representation.
  - **Parameters**
    - `event` - the event about which we want to know the associated proposition.
  - **Returns** - event.toString()

6.2.4 CLASS NotWellFormedChartException

DECLARATION

public class NotWellFormedChartException
extends java.lang.Exception

CONSTRUCTORS

- *NotWellFormedChartException*
  `public` **NotWellFormedChartException**`( )`
- *NotWellFormedChartException*
  `public` **NotWellFormedChartException**`( java.lang.String` **m** `)`

6.2.5 CLASS PivotCTLStarGenerator

This class is a generator for CTL* formulae. The formulae generated are pure-text formulae and do not aim at be used by any model-checker as is.

DECLARATION

public class PivotCTLStarGenerator
extends java.lang.Object
implements ProofGenerator

CONSTRUCTORS

- *PivotCTLStarGenerator*
  `public` **PivotCTLStarGenerator( )**

METHODS

- *generateExistentialProof*
  `public void` **generateExistentialProof(**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart,**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart,**
  `java.util.Collection` **restricted,**
  `be.ac.fundp.info.albert.lsc.verification.EventMapping` **map,**
  `java.io.OutputStream` **out**)
- *generateNoStoryProof*
  `public void` **generateNoStoryProof(**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart,**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart,**
  `java.util.Collection` **restricted,**
  `be.ac.fundp.info.albert.lsc.verification.EventMapping` **map,**
  `java.io.OutputStream` **out**)
- *generateUniversalProof*
  `public void` **generateUniversalProof(**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart,**
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart,**
  `java.util.Collection` **restricted,**
  `be.ac.fundp.info.albert.lsc.verification.EventMapping` **map,**
  `java.io.OutputStream` **out**)
- *toString*
  `public String` **toString( )**

6.2.6 CLASS SimpleEventMapping

DECLARATION

public class SimpleEventMapping
extends java.util.HashMap
implements EventMapping

CONSTRUCTORS

- *SimpleEventMapping*
  `public` **SimpleEventMapping**`( )`

METHODS

- *eventToProp*
  `public String` **eventToProp**`( be.ac.fundp.info.albert.lsc.syntax.LscEvent` **event** `)`
- *put*
  `public Object` **put**`( be.ac.fundp.info.albert.lsc.syntax.LscEvent` **ev**`, java.lang.String` **prop** `)`

6.2.7 CLASS SmvSpecGenerator

This class generates formulae that can be put into the SPEC section of an SMV 2.5.4 input file. It is not compliant with Cadence SMV.

DECLARATION

public class SmvSpecGenerator
extends java.lang.Object
implements ProofGenerator

CONSTRUCTORS

- *SmvSpecGenerator*
  `public` **SmvSpecGenerator**`( )`

METHODS

- *generateExistentialProof*
  `public void` **generateExistentialProof**`(`
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart**`,`
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart**`,`
  `java.util.Collection` **restricted**`,`
  `be.ac.fundp.info.albert.lsc.verification.EventMapping` **map**`,`
  `java.io.OutputStream` **out** `)`
- *generateNoStoryProof*
  `public void` **generateNoStoryProof**`(`
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **prechart**`,`
  `be.ac.fundp.info.albert.lsc.verification.TraceSet` **mainChart**`,`

```
java.util.Collection restricted,
be.ac.fundp.info.albert.lsc.verification.EventMapping map,
java.io.OutputStream out )
```

- *generateUniversalProof*
  ```
  public void generateUniversalProof(
  be.ac.fundp.info.albert.lsc.verification.TraceSet prechart,
  be.ac.fundp.info.albert.lsc.verification.TraceSet mainChart,
  java.util.Collection restricted,
  be.ac.fundp.info.albert.lsc.verification.EventMapping map,
  java.io.OutputStream out )
  ```
- *toString*
  ```
  public String toString( )
  ```

6.2.8 Class **TraceSet**

The automaton recognizing the trace set of a basic chart. It is defined in "Relating intra-object to inter-object specifications", Yves Bontemps, 2001, MSc Thesis, http://www.info.fundp.ac.be/ ybontemp

Declaration

public class TraceSet
extends be.ac.fundp.info.albert.lsc.graph.automaton.Automaton

Constructors

- *TraceSet*
  ```
  public TraceSet( be.ac.fundp.info.albert.lsc.verification.CausalOrder ord,
  be.ac.fundp.info.albert.lsc.syntax.LscBasicChart chart )
  ```
  - **Usage**
    - From a chart and its causal order, build this chart's trace set.
  - **Parameters**
    - `chart` - an LscBasicChart describing the chart of which we want to build the trace set.
    - `ord` - the causal order of chart.

Methods

- *getPivotStates*
  ```
  public Vector getPivotStates( )
  ```
  - **Usage**
    - Computes all the pivots states of this TraceSet.
  - **Returns** - a Vector containing all the pivots states of this TraceSet. if this TraceSet has an empty language, returns null.