COMPOSITIONAL PROOFS

FOR REAL-TIME MODULAR SYSTEMS

BY

CARLO ALBERTO FURIA
Bachelor in Computer Engineering, Politecnico di Milano, 2001

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2003

Chicago, Illinois

Copyright by

Carlo Alberto Furia

2003

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Dino Mandrioli for giving me the opportunity to work on this interesting subject and for his support and valuable experience.

I would also like to thank Matteo Rossi for his help, remarks and suggestions during the preparation of this work: they all have been well addressed, fruitful and indispensable.

I also thank Professor Angelo Morzenti for addressing me to the right literature related to this work, an important step and a source of ideas.

Finally, I thank all the members of the UIC faculty whom I have met during my staying in Chicago, for their roles in the enriching experience this Master has been.

Adhering to the policy "professional acknowledgements only" I am not listing here all the people, friends and relatives, whose personal support has been of the greatest importance for my life as a whole. Instead, I expect to thank them all in more direct ways.

CAF

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

PVS	Prototype Verification System (48)
R/G	Rely/Guarantee (see section 2.1.2.1)
R-T	Real-Time
TCC	Type Correctness Constraint
TD	Time Dependent
TI	Time Independent
TLA	Temporal Logic of Actions (36)
TVS	Trio Verification System (also known as TRIO/PVS)

SUMMARY

One common problem in applying formal methods to the analysis of realistic industrial-size systems is that these methods often do not scale well. In order to overcome such difficulty, formal languages and tools supporting modularization and compositionality must be realized and used.

Under this respect, this thesis addresses the problem of designing techniques and tools to support the formal specification and verification of large modular real-time systems. The reference specification language for this analysis is the temporal metric TRIO.

First, a mapping of the modular features of the TRIO language onto the language of the theorem prover PVS is designed. In connection with this, a number of automated proof strategies are designed to support the conduction of TRIO proofs in the PVS environment.

Second, a rely/guarantee compositional framework for the language TRIO is discussed and a compositional proof rule is derived. This framework is also encoded in the PVS environment, so that it is practically usable.

Finally, the benefits of adopting the proposed rely/guarantee compositional framework are discussed with the aid of working examples.

CHAPTER 1

INTRODUCTION

Today, computer-based time- and safety-critical systems are gaining more and more importance. Examples of such systems are controllers operating in critical environments such as nuclear and chemical plants, supervision systems for flight control tasks, monitoring systems for patients under special treatements. These are usually embedded systems working under critical constraints, or large systems characterized by their complexity. They also often are real-time systems, that is with sharp timing requirements. More precisely, a real-time system (39) is one where the correctness intrinsically depends on the temporal behavior; in other words, the outcomes of the system must be fully time predictable. To put it in another way, while the correctness of a common application only depends on the values it outputs, the correctness of a real-time application depends also on *when* those values are outputed. As a result, the whole development process is a hard, complex and error-prone task.

A common division among real-time systems is between *hard* and *soft* real-time. A *hard* real-time application is one where the failure in meeting the timing requirements may result in totally catastrophic outcomes. Therefore, in a hard real-time system we must absolutely guarantee that all the deadlines are met, otherwise the design is unacceptable. On the other hand, a *soft* real-time application is one where the failure in meeting the timing requirements is undesirable but may be occasionally acceptable. In other words, in soft real-time systems failure in meeting the deadlines has a sustainable, though still highly undesirable, cost.

Formal methods (27), (14) are a valid technique to manage and overcome the difficulties in the development of both hard and soft real-time systems: writing formal unambiguous specifications can guide the traditional activities of validation and testing, and makes the systematic deduction of desired properties of the specified system feasible as early as possible in the development process. To support the use of formal specification languages and to aid the designer, a number of widely used analysis tools has been developed, for example theorem provers (e.g. PVS (48), ACL2 (35)) and model checkers (e.g. SPIN (29), SMV (42)).

However, a common difficulty in applying these methods and tools is that they often do not scale well: they are used with success when dealing with relatively small cases, but they tend to become too difficult to manage if the system under analysis is really large and complex. So, we often face the need of applying formal methods to the analysis of realistic industrial-size systems, while they are usually fit for the use on small "toy" problems only. Consider for example the state explosion problem in model checking or the heavy user interaction required in deductive methods when large, realistic problems are tried. In order to overcome this substantial limitation we need to develop adequate methods to let those basic techniques scale well on large problems, so that they can become more widely used in the development process of critical systems and efficiently help to increase the accuracy and the correctness of the whole process.

1.1 Modularization and compositionality

It is widely acknowledged that an effective way to manage complexity is by means of *modularitazion*: instead of facing the analysis of a large system as a whole, we consider separately the parts into which it is divided and perform local analysis on them. Then, we merge the local results together, to get a global analysis of the whole system. In order to do that, we need specification languages with modularization and object orientation constructs, to compose incrementally the formal descriptions and to reuse them. Moreover, the analysis tools must manage these modular features of the language as well, to support the complexity in the verification process and automate the activity as much as possible. Usually, we refer to the practice of modularization for formal languages and tools as *compositional methods*.

Many of the interesting ideas presented in the literature about compositional methods (reviewed in chapter 2) have been rarely, if ever, concretely applied to the analysis of large systems, as often deplored by the authors themselves. In our opinion, this depends to a certain extent on the fact that the proposed techniques are too often embedded in a rather abstract framework, that requires a noticeable effort to be understood and put into practice. What the average user would require is some form of automation in the process of verification, embedded in a friendly framework. More precisely, the support tools should not only help to carry out the verification of simple, small modules (in-the-small verification), but should furthermore aid the user in the conduction of proofs about global properties of the system, putting together properties of single modules, thus guiding the application of compositional techniques at least in the more commonly occurring cases.

Under these respects, the overall goal of this thesis is to design compositional techniques and tools to support and, as much as possible, automate, the specification and verification of formally specified, large, modular systems. More precisely, the main focus in our work will be on the *composition of modules*, rather than on the decomposition and refinement aspects (as defined in chapter 2). These latter aspects would require a careful analysis of object-oriented constructs and inheritance rules. On the contrary, we want to focus on the scenario where a large modular system has been specified by the user by composing simple modules together. We want to analyze such a system, proving and disproving global properties and checking its consistency and realizability.

1.2 Choice of the reference specification language

Another important preliminary choice we have to make is the formal specification language to use in performing our analysis. Let us briefly consider what are the most desirable features a formal language should have, with respect to the specification and verification of real-time systems.

First, we want it to be not too low-level or requiring a deep expertise to be used. Of course, we know that the proficient use of formal methods necessarily requires a specific training with its theoretical foundational aspects. However, we would like that at least the very basic concepts of our specification language are understandable even by non experts, even if only at an intuitive level. In this case, the domain experts may quickly contribute in the process of writing a formal specification even if their backgrounds do not encompass much logics and computer science.

Second, we want the language to be flexible and expressive enough so that it can be used through all the stages of the development process, stepwise. Under this respect, we do not want it to force the user to introduce details since the very first stages of specification, when one usually wants a certain freedom and does not like to be constrained under one view only.[1]

Third, since our focus is on specifying *large* systems, we want the language to be endowed with modular features that permit the division of the specification into parts and the reuse of modules whenever it is possible.

Finally, we want that the language has (or may be given) an adequate support with tools to automate relevant parts of the specification and verification process. More precisely, we want that the tools guarantee that every detail is in place, while limitating the exposure of the user to the lower-level aspects.

We think that the TRIO specification language (25), (45) fullfills nicely several of the above requirements. TRIO has been created and developed in the *Dipartimento di Elettronica e Informazione* [2] of Politecnico di Milano [3]. It has been, and still is, the preferred reference specification language for the research on formal methods going on in the department. This fact also influenced the choice of the language.

---

[1]As a folkloristic note, we remind that, in the computer jargon, languages enforcing one single narrow view of doing things are nicknamed *bondage-and-discipline* languages (53), (52), as opposite to *liberal* languages.

[2]i.e. Electronics and Computer Science Department; `http://www.elet.polimi.it`

[3]`http://www.polimi.it`

All in all, we choose TRIO as the base reference language for this work. However, we also believe that some of the results that will be obtained can be applied with success to other specification formalisms as well, *mutatis mutandis*.

As discussed above, a very important goal to permit the usability of formal methods on large systems is an adequate support with suitable analysis tools. Currently, a number of analysis tools are available for the TRIO language. In particular, an encoding of the basics (i.e. non modular aspects) of the language is available in PVS (48). Moreover, an integrated environmet offering various verification functionalities, ranging from model checking to theorem proving and test-case generation, is currently being developed. At the moment, an adequate support of the modular features of the language in PVS is lacking; this support should encompass both the mapping of the modular features onto the PVS language, and a support for the computer-aided conduction of proofs.

This work aims at providing such a support. Therefore, we will focus on *deductive* methods, and more precisely we are going to consider *syntactical* techniques, since the PVS encoding of TRIO constitutes a basically syntactical tool[1].

1.3 Goals of the work

After considering all the aforementioned issues, the goal of this thesis is threefold.

- To provide an encoding of the modular features of the TRIO language in PVS, extending the current in-the-small tool. This will consist of both a mapping of the modular

[1] The fact that the encoding of TRIO in PVS (described in chapters 3 and 4) is a *semantical* encoding must not be confused with the fact that the resulting tool is instead a *syntactical* tool.

constructs onto the PVS language and a set of proof strategies to provide an adequate automated support for the conduction of modular proofs.

- To provide a compositionality framework for the language TRIO. More precisely, we are considering how to apply the rely/guarantee paradigm to our specification language, considering the methodological issues that arise and explaining how the results can be concretely employed in PVS encodings of TRIO specifications.
- To analyze and discuss the benefits of adopting the devised rely/guarantee framework in terms of manageability of large systems and proofs, basing our discussion on working examples.

1.4 Structure of the work

This thesis is articulated as follows.

Chapter 2 reviews the recent literature about the topic of compositionality; it also constitutes a general introduction to compositional techniques that are to be employed in the remainder of the work.

Chapter 3 describes the TRIO language and its current in-the-small encoding in PVS.

Chapter 4 introduces a mapping of the modular features of TRIO onto the PVS language; this mapping will be the basis for the conduction of automated proofs with our reference language.

Chapter 5 devises a compositionality framework for the language TRIO. More precisely, we consider how a rely/guarantee specification should be written in TRIO and we prove an inference rule to carry out compositional proofs.

Chapter 6 considers how to effectively help and automate the conduction of proofs in TRIO/PVS. To achieve this, we design a number of PVS proof strategies to be used with the TRIO/PVS tool to facilitate both general modular proofs and rely/guarantee proofs. Moreover, we compare two proofs of the same system done with and without adopting the rely/guarantee methodology and the PVS proof strategies and describe how this affects the complexity of the job.

Chapter 7 draws the most important conclusions to the work and also hints at what its future developments may be.

Finally, appendices A and B list the PVS theories and proof strategies as they have been coded, while appendix C describes the lower-level details of the two PVS proofs considered in chapter 6.

CHAPTER 2

RELATED WORKS ON COMPOSITIONALITY

Nowadays, the urge to overcome the inherent limits of scalability of the most popular formal methods is a common awareness among the research and industrial communities. The usual term by which these techniques, aiming at applying formal languages and methods to large problems, are usually labeled is *compositionality*. Roughly speaking, by this term we mean any method where a large problem is divided into related parts, so that the burden of the analysis of the original task is split into smaller, more manageable subproblems. The method should then allow to verify certain desired global properties by composing intermediate results, provable independently on the parts in which the problem has been divided with traditional in-the-small techniques. The following general introduction to the issue of compositionality follows the survey (16).

2.0.1 The compositional paradigm

Strictly speaking, compositionality is the technical property of a language or a method by which it is possible to verify that a formalized system meets its specification only on the basis of the verified specifications of its constituent components, usually called *modules*, and on how they are combined (composed). To be more formally precise, let us consider a system Π for which we want to prove that some specification property φ holds. If Π can be expressed as a composition of n parts $\Pi_1, \ldots, \Pi_n$ as in $\Pi \equiv \Pi_1 \parallel \ldots \parallel \Pi_n$ where "$\parallel$" denotes some defined form of composition, then a compositional proof of φ can be conducted as follows:

1. Find suitable properties φ_i for each of the parts Π_i, $i = 1, \ldots, n$ such that the next step is possible. The verification of these properties can be performed by means of traditional in-the-small verification techniques if the Π_i are small enough. Otherwise we can further subdivide them into parts and apply the algorithm recursively.

2. Prove that from the fact that each of the Π_i satisfies property φ_i it can be inferred that the whole system Π satisfies φ, i.e.

$$\models \left(\bigwedge_{i=1\ldots n} (\Pi_i \models \varphi_i) \right) \Rightarrow (\Pi \models \varphi)$$

   The technical property of compositionality specifies under which conditions this inference step is sound.

Hence, compositional techniques are yet another application of the *divide et impera* (divide and conquer) paradigm, to the specification and verification of large systems.

2.0.2 Decomposing and composing

Compositional methods can be considered under two complementary aspects with respect to the development process of a system.

The first one is the *decomposition* aspect. The development of a system can start from a very high-level and terse specification, that includes some desired global properties it must respect. The developer refines this initial specification through a series of steps, hierarchically ordered (see for example (18)). Each step produces a lower-level version of the specification, which will be eventually implementable. This is the well-known *top-down* paradigm for the development of programs. By applying formal methods to this process we can ensure that each step in the development is correct with respect to the originally specified properties, so that errors can be identified and corrected as early as possible. This stepwise verification paradigm is usually known as *verify-while-develop*. Well-known formalisms adopting this paradigm are the B method (6) and the Z specification language (58), (60). Moreover, a compositional technique allows the developer to perform such kind of *a priori* refinement process: at each step the higher-level specification is split into parts which describe lower-level details. The correctness of these new details is verified independently and then the global correctness of the composite system formed by these parts is inferred according to compositional rules. Section 2.1.1 below discusses some ideas about this aspect of compositionality found in the recent literature.

The second aspect is instead the *composition* aspect. Not every development process can be performed as discussed in the paragraph above. One often has to deal with a complete specification of a complex system which should be verified for consistency and correctness with respect to certain properties. Such a *a posteriori* verification can still take advantage of the partitioning of the system into parts to reduce the overall complexity of the process. Moreover, such an approach allows the reusability of individually specified and developed components, since they can be combined into larger systems in different manners, in a *bottom-up* style of composition, without the need to re-verify their individual properties every time, but simply relying on compositional proof rules to infer the global properties of the system. Articles about such paradigms for composing individually specified modules are reviewed in section 2.1.2.

To mirror the distinction between these two aspects of compositionality, the terminology has evolved into the term *modularity* to refer to the composition aspect, while using the term *compositionality* in a narrower sense to refer to the decomposition in a top-down refinement process. However, these uses of the vocabulary are neither accepted by all authors, nor used with perfect consistency even by those who adopt them. That is why we will not bother to adhere to a precise distinction, but we will use them both, rather freely. More precisely, hereinafter we will generally use the term *compositionality* to refer to the technical property in use, while referring to the parts in which a system is decomposed as *modules*.

2.0.3 The origins of compositionality

The principle of compositionality was first formulated within the context of propositional logic and philosophy of language by Gottlob Frege (23). In a broad sense, a compositional system is a system where the global properties are function of the properties of the parts in which the system is divided. Most natural and artificial artifacts intuitively exhibit this property. Natural language is compositional as well, since the meaning of a sentence in English depends on the meaning of its parts.[1]

---

[1] Contemporary linguists tend to consider the principle of compositionality valid in a narrower sense, that is when atomic representations make the same semantic contribution in every context in which they occur. In this sense, natural language is not fully compositional, since the meaning of a noun or of a verb is not completely independent of its context: a typical example of this is the different meaning of the verb "to kick" in the sentences "he kicked the ball" and "he kicked the bucket". However, the compositionality principle is commonly used in its original, broader sense.

It is interesting to note that several formal techniques for the analysis of computer programs have evolved from an initial non-compositional stage to a structured compositional one, as the programs under analysis become more complex. For example, for sequential programs, the first non-compositional method was the one by Floyd (22); two years later a new axiomatic compositional technique was proposed by Hoare (28). It is no surprise that a similar evolution is happening in the analysis of real-time systems as they grow in importance. What we are seeking is to extend the structured paradigm for program development to the analysis of real-time concurrent systems.

2.1 Compositional techniques for the verification of real-time systems

2.1.1 Decomposition of systems

The first framework for applying compositionality to the decomposition of high-level specifications into lower level ones is proposed by Abadi and Lamport (4). Let us consider a system or a program specified using TLA formulae (36). If M generically indicates the complete system, we want to find a refinement, i.e. a lower-level description, M^l of it which implements M, that is such that every behavior of the world which satisfies M^l also satisfies M; in formulae: $M^l \Rightarrow M$. In order to prove that, one has to find a *refinement mapping* (1), that is, roughly speaking, a suitable function which maps the state variables of M^l into state variables of M so that any formula true for M^l is also true for M.

However, being M in general a complex system, finding such a mapping may be impractical. What we want to do is to exploit modularization to render the process easier. If $M_1, \ldots, M_n$ are TLA formulae representing the n parts into which M is split, then their conjunction represents the whole system M: $M \equiv M_1 \wedge \cdots \wedge M_n$. In fact, every possible history of the universe is compatible with M if and only if it is compatible with all the $M_i, i = 1, \ldots, n$.

Let us now refine each of these modules individually into lower-level descriptions $M_1^l, \ldots, M_n^l$. Our original proof that $M^l \Rightarrow M$ is now equivalent to $M_1^l \wedge \cdots \wedge M_n^l \Rightarrow M_1 \wedge \cdots \wedge M_n$. We would like to decompose this proof into the presumably simpler proofs $M_i^l \Rightarrow M_i$ for $i = 1, \ldots, n$. In general, this reduction requires some additional assumptions about the modules M_i, to assure the soundness of the reasoning. We usually refer to these additional assumptions as E_i. If these assumptions are carefully chosen, a Decomposition Theorem holds, that assures the soundness of the decomposition process. More precisely, but avoiding any technical detail of the TLA formalism, we may roughly say that the theorem has as hypotheses:

1. That the conjunction of the higher-level specifications M_i satisfies each of the assumptions E_i, modulo some additional technical details
2. That $E_i \wedge M_i^l \Rightarrow M_i$ for each $i = 1, \ldots, n$, that is that the modules individually implements the higher-level specifications, with some additional technical assumptions.

These hypotheses let us conclude the desired result that $M_1^l \wedge \cdots \wedge M_n^l \Rightarrow M_1 \wedge \cdots \wedge M_n$ that is $M^l \Rightarrow M$. The assumptions E_i are all similar to those used in what is called the *rely/guarantee* paradigm, discussed in more detail below (see section 2.1.2.1) in the context of composition of open systems. In fact, the aforementioned Decomposition Theorem can be reduced to a consequence of the analogous, but more powerful, Composition Theorem.

The framework proposed by Abadi and Lamport naturally refers to a discrete temporal model of the world with states and transitions. The language TLA is rather low-level and requires the user to deal with a number of technicalities. Moreover, the approach is an abstract one, strongly modeled after purely mathematical formalisms. A substantial practice would be required for any user, before being able to put into practice some of the principles or to translate informal specifications into a canonical formula of this framework. For what concerns machine support for the conduction of proofs, the basic of the logic is low-level enough so that it could be implemented fairly simply into any theorem prover. However, the user would still be completely responsible for the management of all the compositional details, along with the main choices for the conduction of the proofs.

Hooman (30) proposes a framework to support the top-down design of real-time systems based on logical formulae at the semantic level. This very simple formalism has been implemented in the language of PVS to have some mechanized support to the conduction of proofs. The approach explicitly wants to be as general as possible with respect to both language-dependent implementation choices and the time model (i.e. discrete or continuous).

The basic primitive to express properties of a system is the *observation function*, a function from the time domain to a set of events from Ev:

$$ObsFunct : Time \rightarrow 2^{Ev}$$

The values assumed by these functions completely describe the temporal behavior of a module. If we have two or more modules, we define their parallel composition as a module completely described by the pointwise union of the composed observation functions, over a set of events given by the union of the set of events of each function. To guarantee consistency, we must also add the restriction that the observation functions of the various modules are equal on the events that are in more than one event set. So, if a module M_1 is described by an observation function obs_1 over a set of events Ev_1, and similarly for a module M_2, their parallel composition defines the module M, described by the observation function over:

$$M \equiv M_1 \parallel M_2 : \qquad obs : Time \rightarrow 2^{Ev_1 \cup Ev_2}$$

and assuming values:

$$evt \in obs(t) \quad \Leftrightarrow \quad \left(\begin{array}{l} \left(\begin{array}{l} evt \in Ev_1 \cap Ev_2 \\ \qquad \wedge \\ \forall u \in Time(evt \in obs_1(u) \Leftrightarrow evt \in obs_2(u)) \end{array} \right) \\ \qquad \vee \\ \left(\begin{array}{l} evt \in Ev_1 \wedge evt \notin Ev_2 \\ \qquad \vee \\ evt \notin Ev_2 \wedge evt \in Ev_2 \end{array} \right) \end{array} \right)$$

Hooman defines this notion of parallel composition in the higher-order logic of PVS and shows that alternative but equivalent definitions are possible.

Tightly connected with the notion of observation function is the notion of *specification* of a component. This is just an observation function for a set of events Ev characterized by a certain property P:

$$spec(Ev, P): \qquad obs : Time \rightarrow 2^{Ev} \quad \text{s.t. } P(obs(t))\ \forall t \in Time$$

With these definitions, we are now ready to formulate a compositional proof rule, that basically states under which conditions the parallel composition of two modules with given specifications implements a module with a global property given by the conjunction of the specification properties of the two modules. If the two specifications are given as $spec(Ev_1, P_1)$ and $spec(Ev_2, P_2)$ we simply require that the validity of P_1 only depends on events in Ev_1 and similarly for P_2. We denote this fact with predicates of the form $OnlyDep(P_1, Ev_1)$. Under these assumption, the following *Compositional Rule* holds:

$$\begin{gathered} OnlyDep(P_1, Ev_1) \wedge OnlyDep(P_2, Ev_2) \\ \Rightarrow \\ spec(Ev_1, P_1) \parallel spec(Ev_2, P_2) \sqsubseteq spec(Ev_1 \cup Ev_2, P_1 \wedge P_2) \end{gathered}$$

where $\sqsubseteq$ indicates the refinement relation between modules, basically reducible to a logical implication between properties of specifications, plus some technical details, omitted here. Again, alternative semantic definitions for parallel compositions are proposed and shown to be equivalent to the previous one.

The author also analyzes the *Hiding* operation, another way to modify a module in a refinement process. This basically consists in removing from an observation function a number of elements of its event set. A sound inference rule for modules obtained by hiding events is shown.

Finally, a system described by both discrete and continuous events is specified and analyzed according to the given definitions for observation functions.

Hooman's proposal for a framework is indeed very general and basic. It is probably even too general and basic to be applied to large system analysis where a purely semantic description is too low level to be used in early phases of the specification process. It may be possible to extend and add more expressiveness to this framework, in particular by allowing more complex notions of composition and hiding, and by using a richer and more intuitive language to express properties of a system. The simplicity of the underlying structure of the framework may be instead an advantage in providing an implementation into an automated proof checker like PVS.

Olderog and Dierks (47) propose a paradigm to decompose specifications of real-time applications so that time-critical aspects are confined in some modules only.

Starting with a specification written in Duration Calculus (13), or more precisely into a subset of it that is implementable into executable timed programs, it is always possible to decompose the specification into two parts, such that one is a completely untimed version of the system, while the other is a pure timer. By adequate communication, the composition of these two modules is an implementation of the original real-time system, that is exhibits the same temporal behavior. The kind of composition considered here is obtained by asynchronous communications between modules so that output of one module serves as input of the other and vice-versa. A general algorithm to perform automatically this decomposition is discussed and shown on an example.

The work by Olderog and Dierks, even if tightly modeled after a single formal language of specification (Predicate Calculus), could also be regarded to as a general guideline in writing specifications of real-time applications to facilitate a formal analysis. In other words, a specification written with a sharp division between temporal and non-temporal aspects may be useful to allow some form of automated analysis. Of course, in general it is rather difficult and counterintuitive to write a specification with this division in mind from scratch, so it is higly desireable to have a general decomposition algorithm to automatically do so.

2.1.2 Composition of systems

2.1.2.1 The rely/guarantee paradigm

An independently specified module is in general an *open* system, that is a system interacting with an external environment which provides its inputs. When proving properties of an open system, we would generally want to express them independent of the possible behavior of the outside world interacting with the component. This would guarantee the highest reusability of the component, which could then be plugged into any system without changing its behavior.

However, this is in general not possible, since it is often the case that a component behaves correctly only if the environment does the same. For example, the environment providing the inputs may be required to adhere to a certain communication protocol or, in the context of digital circuits, to input a nominal voltage value, with some fixed tolerance, but without intermediate acceptable values.

In order to deal with this issue, a solution was first proposed by Misra and Chandy (43) for synchronous communications and shortly after by Jones (31), (32) for shared variable concurrency. They proposed what goes under the name of *rely/guarantee* paradigm, according to the terminology introduced by Jones. Other names for basically the same paradigm are *assumption/commitment* (Misra and Chandy) or *assumption/guarantee.* The rely/guarantee paradigm for writing the specification of a module simply consists in making explicit assumptions about the behavior of the environment under which the module behaves as specified. In other words, the specification of the component is of the form: *assuming* the environment to behave as E, we can *guarantee* that the module exhibits property M. It is advisable to choose E to be the minimal assumption on the environment to guarantee the behavior M, to permit the maximum reusability of the component. Moreover, it is important that the environment property E does not refer in any manner to implementation details of the environment, considered as a black-box. It is interesting to note that the rely/guarantee paradigm can be considered as a generalization of the well-known precondition/postcondition style for specifications of sequential programs. Frameworks based on the rely/guarantee paradigm are discussed in the following paragraphs.

Abadi and Lamport conduct an in-depth analysis of the many technical details that should be taken into account when considering composition of modules specified under the rely/guarantee paradigm. (4) considers the problem strictly connected with the decomposition aspect, in the context of TLA (36) as a specification language. The same authors in (2) use a more complicated semantic model with agents to analyze the same problem from an even more theoretical point of view. Finally, Abadi and Merz (5) discuss rely/guarantee composition with reference to the logics TLA and CTL* and derive some proof rules for these logics, in the spirit of the other two papers. We discuss these three related works together, with main reference to (4), since it is probably the most general and most self-contained of them, at least with respect to our interests.

According to the proposal of Abadi and Lamport, we describe the rely/guarantee paradigm for the specification of an open system, considering an underlying discrete temporal model with states and transitions. A rely/guarantee specification of an open system Π should be written as a formula of the form $E \stackrel{+}{\rhd} M$ where the derived TLA operator $\stackrel{+}{\rhd}$ means that M holds at least one step longer than E does or, in other words, that M becomes false only *after* E has become false. More precisely, if we consider a history of the system (also called a behavior), $E \stackrel{+}{\rhd} M$ is true if and only if $E \Rightarrow M$ is true and, for every $n \geq 0$, if E holds for the first n states, then M holds for the first $n + 1$ states. This is the best way to describe a rely/guarantee assumption, because it states exactly that:

1. the system guarantees an expected behavior, provided the environment evolves as modeled
2. if the environment stops following its model, the system may behave incorrectly. However, it does that only one time step after the failure of the environment.

This corresponds to our intuitive notion of rely/guarantee paradigm and also has the additional advantage that it leads to simpler rules for composition of modules.

Let us now consider two systems Π_1 and Π_2 whose rely/guarantee specifications are $E_1 \overset{+}{\Rightarrow} M_1$ and $E_2 \overset{+}{\Rightarrow} M_2$, respectively. In the simple case that $E_1 \equiv M_2$ and $E_2 \equiv M_1$ one would expect to conclude that the composite system Π exhibits property $M_1 \wedge M_2$ without any additional assumption, since each module satisfies the other module's environment assumption. However, this conclusion is not valid in the general case, but depends on the kind of environment assumptions we have: if they are safety properties or not. A safety property, as defined in (7), (57), is one that is finitely refutable, i.e. a violation at a single finite instant of time suffices to make it false. With safety properties as environment assumptions, compositional reasoning has simpler rules. In order to deal with general environment properties, we have to define the safety closure $\mathcal{C}(F)$ of any TLA formula F, as the strongest safety property such that $\models F \Rightarrow \mathcal{C}(F)$.

We are now ready to formulate a general Composition Theorem, that states a sound inference rule to manage the composition of modules specified under the rely/guarantee paradigm. Let us consider n modules, each with a specification of the form $E_i \overset{+}{\Rightarrow} M_i$ for $i = 1, \ldots, n$. We want to prove that the composition (i.e. conjunction in this approach) of the modules satisfies a global rely/guarantee specification $E \overset{+}{\Rightarrow} M$. The hypotheses to the theorem are roughly the following (we are still omitting a number of technical details):

1. The closure of the global environment assumption $\mathcal{C}(E)$ and the conjunction of the closures of the modules' properties $\mathcal{C}(M_i)$ imply all the environment assumptions E_i and the closure of the global property $\mathcal{C}(M)$.

2. The global environment assumption E and the conjunction of the modules' properties M_i imply the global property M.

Under these assumptions, we can conclude the soundness of the global specification, that is:

$$\models \bigwedge_{i=1,\ldots,n} (E_i \stackrel{+}{\rhd} M_i) \Rightarrow (E \stackrel{+}{\rhd} M)$$

This rather general result is treated with even more technical considerations in a more complex semantic model in (2). A normal form for a specification in the rely/guarantee style is proposed, with restrictions to deal with issues like partial program specifications, machine-realizability and safety closures. In particular, it is shown how the non-safety part of the environent assumption can be pushed into the M part of the model, so that the Composition Theorem can be stated in a simpler way. However, this interesting consideration is instead more of intellectual interest than of practical use, as also remarked by the authors themselves in (4), since it would be highly innatural to write a specification in that constrained form.

Abadi and Merz get to a similar Composition Theorem (5), adopting the temporal logics TLA and CTL*. They first briefly study the concept of composition in a rather general abstract logic framework, then they infer the compositional rules for the chosen languages. It is likely that such an approach could be extended to other temporal logic formalisms as well, under the same general paradigm.

The proposal by Abadi and Lamport nicely points out some fundamental facts we must deal with whenever conducting proofs in a compositional framework. A concrete application of such principles would still require a considerable effort in order to make all the complex technical details as hidden as possible from the user, allowing an approach to the specification process as smooth as possible. The authors point out that the Composition Theorem has only been applied to toy examples, and propose an approach based on mechanized verification for larger system, using the Composition Theorem to divide the labor. To put into practice this suggested direction of work would definitely be an interesting achievement.

Namjoshi and Trefler (46) consider the compositional inference rule proposed by Abadi and Lamport in (4), together with similar ones, in order to analyze them with respect to the problem of completeness. In fact, while all the proposed compositional proof rules are sound rules, that is they infer true facts from true premises, many of them show to be *incomplete*, that is there are true properties of the global system which cannot be proved using those inference rules. More precisely, this is the case with the rule in (4) and with other similar rules exploiting circular reasoning. By circular rules, we mean proof rules where the guarantee of a module is constrained not only by the assumption of the same module but also by the guarantee of the other modules. A noticeable fact is that the counterexamples used to show the incompleteness are neither particularly complex nor involve large systems: this suggests that the incompleteness of those rules may well be a practical problem in the verification of systems, and not just a theoretical nuisance.

However, modifying the available inference rules to make them complete is possible and does not require to sacrifice substantially the formal simplicity of the rules. In particular, it is not necessary to use non-circular rules instead of circular ones: another result drawn by the authors is that circular and non-circular proof rules can be equivalent and it is always possible to pass from a circular description of a modular system to a non-circular one. What can be done to obtain complete inference rules is to strengthen the hypotheses by adding some form of auxiliary assertion, i.e. an additional property that strengthen the guarantee part of each module. The effort needed to find those additional assertions may affect the complexity of the proof and require substantial user interaction. The authors precisely formulate a complete proof rule, using the formalism of communicating processes and common linear temporal logic. The most relevant aspect is the fact that there is a sort of trade-off between the ease of applicability of a compositional proof rule and the completeness of such a rule: simpler rules do not require additional intermediate assertions, which are often non-trivial to be formulated, but they do not allow some true facts to be proven; conversely, more complex rules are complete but are more difficult to be applied automatically.

2.1.2.2 The lazy approach

The rely/guarantee approach to the specification of open systems is the most studied and analyzed in the literature about compositional verification. However, practical difficulties have prevented it from being concretely and widely used on large cases. The biggest of such difficulties is probably the fact that the compositional inference rules for this paradigm require that the assumptions about the environment of each module are subsumed by the specifications of the other modules of the system providing inputs to the first module. This fact is often a practical problem because it forces the specification to be detailed enough to discharge these constraints since the very first stages of the formal analysis. This in turn requires to anticipate a number of implementation details that would not pertain at all to the initial specification phases and are also too complex to be considered there.

Shankar proposes an alternate framework for the study of compositionality with open systems, called the *lazy* approach (55). In lazy composition, if we want to prove that a certain component Π_1 has a property M_1 under certain environment assumptions, we prove M_1 to be valid for the composite system $\Pi_1 \parallel E_1$ obtained by composing Π_1 with an abstract environment specification E_1 which captures the expected behavior of the environment. Later, the component Π_1 will be eventually composed with another module, say Π_2, to form a system $\Pi_1 \parallel \Pi_2$. In order to guarantee that the property M_1 is still valid for the global system, the lazy paradigm considers the modified global system $\Pi \triangleq \Pi_1 \parallel (\Pi_2 \wedge E_1)$. M_1 surely holds on Π, since the environment behavior E_1 is made explicit part of the model. Later, during the following refinements of the system specification, we will have to show that $\Pi_1 \parallel (\Pi_2 \wedge E_1)$ is refined by $\Pi_1 \parallel \Pi_2$ so that the environment assumptions will be correctly discharged. This later, lazy analysis of the environment specifications does not force the model to be too detailed since the first stages, but permits the user to care about environment assumptions only at the end of the specification process, when enough implementation details have naturally come into the picture. The lazy compositional approach seems general enough to be applied to a variety of formalisms and specification models. However, Shankar details the analysis with respect to a discrete-time model of computation, the asynchronous transition systems.

Let us briefly analyze the main differences and similarities between the rely/guarantee and the lazy approaches.

- Rely/guarantee verification does not allow the later use of any implementation detail of a component to discharge the environment assumptions, but requires to anticipate all those component properties needed to satisfy the environment constraints. The lazy approach allows instead to discharge those constraints lazily as the specification is refined, hence using implementation details when they are available.
- The rely/guarantee proof rules as proposed in (4) consider composition as conjunction. This is compatible with the notion of composition for most formalisms, but it may be the case that it is needed to consider different kinds of composition, that do not fit under the idea of conjunction. The lazy approach does not consider any specific paradigm of composition, but its inference rules are compatible with whatever notion is adopted by the chosen formal language.
- The accumulation of environment assumptions done in the lazy compositional verifications fits well with the fact that specifications are often partial and not so strong during the initial phases of analysis, so that we do not have to care about details of consistency during those phases. On the other hand, this approach does not allow each component

to be refined independently from the others, since this is possible only when its whole specification already subsumes all the environment constraints attached to that module.

- The lazy compositional approach may yield inconsistent specifications, since it may be impossible to refine a component so that it matches all the required environment assumptions. This is a price to pay to have more freedom during the initial stages of analysis.

Shankar also proves a number of proof rules that allow one to deduce that a component of the form $\Pi_1 \parallel \Pi_2 \wedge E_1$ is refined by the simpler component $\Pi_1 \parallel \Pi_2$. These proof rules are tailored to the asynchronous state transition formalism but, once all the language details are removed, they show to be general and simple rules to deduce when a global property φ is preserved under the refinement. It is interesting to note that also with this approach, liveness properties are harder to handle than safety properties, so that stronger hypotheses must be considered. We omit here the details about these issues.

Shankar proposes a different and new approach to the compositional verification of modular open systems. The approach has a number of advantages over the more used rely/guarantee paradigm, and a number of disadvantages as well. In view of an implementation of a lazy compositional mechanism into an automated analysis tool, a noticeably advantageous aspect is that the lazy approach requires no *ad hoc* machinery since it relies on existing techniques for proving refinement properties only. On the other hand, it is still to be understood if the approach can be effectively used on really large system, without the specification becoming too complex to tackle.

Finkbeiner, Manna and Sipma (21) propose a formal framework for the analysis and verification of modular systems, modeled as fair transition systems (40), with linear temporal logic as specification language to express properties. To solve the problem of discharging environment assumptions when composing a number of open modules, they propose a paradigm similar to the lazy approach proposed by Shankar (see the previous paragraph), where if an assumption cannot be discharged, it is simply made part of the composite model and eventually discharged later, by implementation choices. However, in this new framework, it is also possible to discharge immediately the assumptions by means of properties of other composed components, similarly in this sense to the rely/guarantee approach. The general technique tries to avoid anticipating any assumption about a module made by other modules, and guarantees that properties are preserved under parallel composition.

Every module is formalized as a fair transition system, which basically consists of an interface and a body. The body defines the private parts of the module and its internal behavior; the interface is instead the public part of the module and consists of a number of variables (representing state values or state functions) and of transitions. Different modules can be combined together and modified to form a larger system, by means of the following operations:

Parallel composition, which merges two modules into one, merging the interfaces and synchronizing transitions with the same name, while guaranteeing the non interference of the private parts of the two modules.

Hiding, which removes a number of variables or transitions from a module's interface.

Renaming, which renames variables or transitions of the interface.

Augmenting, which adds new variables to the interface of the module; they assume values specified as an expression of other public and private variables of the module.

Restricting, which replaces variables of the interface by expressions over other variables only.

In order to guarantee that the application of these operation does not yield inconsistent specifications, a number of compatibility conditions between modules are formulated and discussed with some detail, together with the precise semantics of the above operations. Compositionality really comes into the picture when a number of *property inheritance rules* are shown. A property inheritance rule states under which conditions a property valid for a module Π is still valid when the module is modified with one of the aforementioned operations. In the spirit of lazy composition, no specific assumptions are made on the module's environment before the application of the operation. On the contrary, the required assumptions are carried over the composite system and can be discharged later when it becomes possible. Another advantage of this approach is that these compositional rules are rather simple in their forms, since all the details about the nature of the environmental assumptions are not explicit part of the verification paradigm. The soundness of the given inference rules can be proved, by showing that a refinement mapping (1) exists between the original module and the one modified after the application of the operation.

Another interesting discussed issue is module abstraction. They show an inference rule which makes it possible to replace a module with an internally simpler one with the same external behavior. This justifies the use of higher-level abstractions to replace a part of a large system, in order to focus the analysis on the other parts of the system and avoid any insignificant detail. Another extension of the ideas about property inheritance is done with a proof rule that handles recursive definitions of composite modules and applies an induction principle to derive global properties. This rule is rather simple and intuitive, and is applied to an example of recursively defined system.

Finkbeiner, Manna and Sipma propose a rather liberal approach to the verification of modular systems, where assumptions to guarantee results are generated naturally in the course of the proof, and whose correctness is proven lazily when possible during later phases of the specification process, by refinement or by properties of other modules composed together. As often the case with these abstract frameworks, a considerable effort need be made in order to implement these ideas with suitable analysis tools, extend the framework to other formalisms and hide most details of the proofs to the user. In particular, we point out that this approach would require some form of automatic high-level generation of intermediate assertions during the conduction of a proof (8).

Bjørner, Manna, Sipma and Uribe (9) propose a framework similar to the one in (21) and consider how to provide it with some form of automation in the conduction of proofs, by exploiting the capabilities of the tool STeP (11), (10). In what is basically an evolution of the previously seen framework (21) and of the similar one (12) to describe real-time systems, they adopt the computational model of clocked transition systems (41) with a dense time model. As usual, properties of the systems are expressed with linear temporal logic formulae.

The resulting framework allows the straightforward extension of the aforementioned operations of parallel composition, hiding, renaming, etc., to modules of a real-time system. This also allows the reuse of some of the compositional inference rules seen before in this new, broader context. Moreover, the authors point out some additional facts we must consider when modeling a real-time system in a continuous time framework. In particular, a fundamental requirement a system must satisfy in order to be implementable is that it must exhibit a *non-Zeno* behavior. This requirement can be expressed in several different ways (3), (24); for now it suffices to say that it means that each variable of the system can change its value only a finite number of times in any finite time interval. Non-Zenoness is not preserved under composition of modules; however there is a closely related condition called *receptiveness* that implies non-Zenoness and is preserved under composition. This notion is discussed in the context of composite clocked transition systems. Finally, it is shown with a significative example how the ideas introduced before can be implemented into the STeP system to provide a support for formal analysis. In particular, it is shown how the tool can aid the verification of non-Zenoness properties and of safety properties, by exploiting the capabilities STeP has to generate invariants in a given transition system.

The work by Bjørner, Manna, Sipma and Uribe is one of the first attempts to really put into practice some of the interesting ideas about compositional reasoning found in the literature. In particular, we point out once again the importance of an adequate machine support in the conduction of proofs not only in-the-small, where a number of tools is already available and automated approaches like model-checking are viable, but also in-the-large, where compositional techniques must come into play.

2.1.3 Compositionality in non-deductive frameworks

The compositional paradigm is general enough so that it can be applied to a variety of formal languages. Our main interests are for deductive frameworks, that is where verification of global properties is done by finding out formal proofs of those properties in the chosen language.

A different issue is how to carry out the verification of the properties that are local to each base module. A possibility is to use algorithmic model checking techniques (15), (38) to automatically verify those properties, thus using deductive proofs only to compose local properties into global ones. This is suitable whenever the finite state model used in model checking is appliable locally. On the other hand, we can also carry out the entire verification of the system in a deductive framework, without any fully automatic technique. This permits the in-the-small verification also of those subsytems where the finite state model is not appliable. Of course, we can also have the best of both worlds, by choosing which technique to adopt for the verification of local properties on a per case basis. A really powerful and complete framework should encompass both ways, and it should be supported by a tool suite to perform such a flexible approach to the verification of large systems.

Even if our main interests in this work are for a deductive framework, in this section we very briefly hint at some articles found in the literature where compositionality is considered from a different perspective, that is in a fully state-based model checking framework. This means that these approaches consider the problem of devising automated algorithmic techniques that scale well on large state-based systems.

In order to allow global properties verification a model checker should first of all have a way to represent modules and their composition in a natural manner. Moreover, it should allow the verification of properties of single modules with respect to any possible behavior of its environment. The behavior of the environment should be expressible both as temporal logic formulae and as state-based machines. Finally, it should allow the development of a system both in a top-down and in a bottom-up process similarly to what discussed above with reference to deductive frameworks. (26) discusses some of these issues in depth and formulates a number of algorithms and techniques. The reference temporal logic is CTL* and the results are of rather general applicability.

Another approach is the one proposed with reference to the widely used modeling formalism of Petri nets. These other techniques consider compositional verification based on condensation rules, that is rules for simplification and abstraction of subsystems of a given modular net. These techniques allow a significant reduction of the overall complexity in the verification of global properties. In particular, (34) considers compositional condensation rules for asynchronous processes and heuristic techniques for large concurrent systems, with particular emphasis on the state reachability problem. Similar techniques are discussed, together with many others, in a more general framework in (33), where a wide range of problems is considered. We point out that these approaches, based on condensation rules, are really compositional techniques, unlike other rule-based reduction schemas (56), (19), even if they both can be applied to the same class of problems.

2.1.4 Another use of compositionality in system analysis

In this section we make a brief detour from the topic of compositional proofs to review a couple of papers that consider different uses of a formal modular specification of a system other than deductive reasoning. We believe this related topic may share some basic ideas with the deductive approach.

Morzenti, Morasca and San Pietro (44), (54) propose methods for the systematic generation of execution sequences from the formal modular specification of a real-time system. These execution sequences can be used to automate an activity of functional testing, that is an extensive testing of the system driven by system requirements formalized during the specification phase. A modular system is specified with the temporal metric TRIO (25), (45), even if the framework applies to other temporal-logic formalisms as well. We also suppose to have working algorithms for the generation of execution sequences for simple modules, that is non-composite basic modules.

In order to exploit the topological information associated with a modular specification, a connection graph is built to represent such information. Every simple module is represented by a node in the graph; arcs link connected modules of the specification, with direction going from the module providing the output to the module using it as input. Two different algorithms are presented, one for acyclic graphs and the other for the general case of cyclic graphs. The two algorithms basically exploit the topological ordering of nodes induced by the graph in order to systematically generate execution sequences. User guidance is then needed to choose the desired degree of adequacy of the generated execution sequences. Correctness and time complexity of the given algorithms are also analyzed. In particular it is shown which properties of regularity of the specification must be assumed in order to guarantee the termination and convergence of the given procedures. In order to apply the same algorithms to arbitrarily large and complex modular specifications, graph theoretical techniques are discussed to manipulate and rearrange the connection graph into a simpler but equivalent one. How to deal efficiently with specifications with many hierarchical levels of encapsulation is also discussed. Finally, a prototype tool to perform the automated analysis is implemented and shown on a case study, with interesting results.

We think that the idea of representing the topological structure of a composite system with a graph may aid the analysis of strategies for proof conduction, by exploiting possible implicit cause/effect relations between modules. The possibility of applying a large variety of graph analysis techniques to the representation may lead to interesting results even on really large and complicated systems. Of course, this paradigm must be adapted to the case of compositional deductive reasoning, requiring a number of substantial changes.

2.2 The complexity of compositional techniques

A primary concern for the concrete applicability of the methods for the analysis of large systems is that their complexity is as small as possible, as neatly pointed out by Dijkstra (17) in 1969:

> On a number of occasions I have stated the requirements that if we ever want to be able to compose really large programs reliably, we need a discipline such that the intellectual effort E (measured in some loose sense) needed to understand a program does not grow more rapidly than proportional to the program length L (measured in an equally loose sense) and that if the best we can attain is a growth of E proportional to, say, L^2, we had better admit defeat. As an aside I used to express my fear that many programs were written in such a fashion that the functional dependence was more like an exponential growth!

Unfortunately, compositional specification methods have a worst-case complexity that depends exponentially on the number of modules we are considering. This happens in the most general case, that is when we allow any possible kind of composition between modules, and we want to analyze all the possible reached states of the resulting global system. In order to make this consideration more concrete, let us consider a common application of modular techniques: modular model checking with the rely/guarantee paradigm. It has been shown in (59) that this problem is *EXPSPACE-complete*, so that it is inherently intractable. Lamport (37) points out some other negative facts about compositional theorem proving. First of all, the application of decomposition proof rules usually does not shorten the length of a proof, but just changes its high-level structure, thus rearranging its lower-level steps, and even adds extra work to handle environment specifications or liveness properties. Second, he points out that the empirical laws that seem to govern the complexity of compositional proving suggest that the length of a proof is quadratic in the length of the low-level specification, which is an unacceptable result. If we add the fact that deductive verification is furthermore a *semi-decidible* problem, the situation seems definitely hopeless for modular composition.

However, this is fortunately one of the cases where the worst-case scenario happens rather rarely in practice, while the average-case shows a much lower complexity. Submodules of a complex system usually exhibit an interaction which is not too tight, in the sense that only a small part of their variables is in direct relation. We usually identify the public parts of a module that interact with other components as the *interface*. Processes' interaction is recorded by observing the changes occurring in the modules' interfaces and not the internal quantities. It is often the case that the items in the interface are a small part of the total of a module. This usually implies that there is a number of externally visible changes that is not too big, with respect to the number of internal states corresponding to the same observed interface, so that the total number of interesting combinations does not grow too fast when composing modules. These assumptions are also supported by the fact that composite systems of interest are artifacts invented by human beings, and human beings cannot manage an exponential complexity, so that these systems must be effectively describable by a not too high number of modules interacting rather loosely. As a result, compositional techniques applied to the analysis of real systems usually show a complexity that is *linear* in the number of subsystems. Cases of higher complexity may happen in practice but are rare and usually pertain to small parts of the global system only.

All in all, compositional reasoning is the main tool we have to seriously tackle complexity, since it makes reduction to smaller problems and abstraction work together in an effective way.

We have briefly reviewed and summarized a significant part of the literature about compositional techniques for the conduction of proofs. A number of ideas and paradigms from these articles are interesting and are going to be applied to our particular framework. Other aspects of the problem are instead scarcely represented in the current literature, so that our work should cover some of these other issues.

CHAPTER 3

THE TRIO SPECIFICATION LANGUAGE

This chapter presents the TRIO specification language (25), (45) and its current encoding in the PVS theorem prover. More specifically, section 3.1 considers the basic non-modular constructs of the language and their encodings in PVS, while section 3.2 introduces the reader to the modular and object-oriented features of the language.

3.1 TRIO and its encoding in PVS

3.1.1 TRIO in-the-small

TRIO is a typed, linear, metric temporal logic enhanced with object-oriented and modular features, for writing specifications of complex systems. Whenever a distinction is needed, we will refer to the basic aspects of the language, that is the syntax and semantics of formulae predicating about time-dependent and time-independent items, as "TRIO in-the-small", while referring to the remainder of the language as "modular features of TRIO". Following this division, this section describes the former group of features and section 3.2 refers to the latter one.

The truth value of each TRIO formula is given with respect to a current time instant which is left implicit[1]. The basic temporal operator is called *Dist* and relates the current implicit time instant to another time instant. For example, the formula $Dist(F, t)$ where F is a time-

[1] In fact, the name TRIO stands for "Tempo Reale ImplicitO", Italian for "implicit real-time"

dependent formula and t a time distance, means that F holds at a time instant which is t time units from the current one.

Together with this basic temporal operator, TRIO allows the use of all common propositional operators and quantifiers of first-order logic. Combining these with the $Dist$ operator, we define a number of *derived* temporal operators, that naturally express common time relationships. Table I lists the formal definition of all derived TRIO operators: F and G are any time-dependent formulae and t is a time distance. Moreover, note that each standard operator can have a modified version with explicit inclusion/exclusion of time bounds. For example, the definition of $Lasts_{ei}(F, t)$ is $\forall d(0 < d \leq t \rightarrow Dist(F, d)))$.

In TRIO, we call *items* the primitive entities used to represent the system under specification. Among them we have values, predicates, functions, events and states. We distinguish between time-dependent (TD) items, whose value varies over time, and time-independent (TI) items, whose value does not. In particular, events and states are a specialization of time-dependent predicates useful to represent a system in an operational way. An *event* models instantaneous facts, such as the pushing of a button or a change of state. A generic time-dependent predicate E represents an event if it obeys the following behavior (24):

$$UpToNow(\neg E) \wedge NowOn(\neg E)$$

Conversely, a *state* models Boolean values that hold over time intervals and whose transitions are pointwise (i.e. they are events); for example, the value held by a flip-flop device may be

Operator	Definition
$Past(F,t)$	$t > 0 \land Dist(F,-t)$
$Futr(F,t)$	$t > 0 \land Dist(F,t)$
$Som(F)$	$\exists d Dist(F,d)$
$Alw(F)$	$\neg Som(\neg F)$
$SomP(F)$	$\exists d(d > 0 \land Dist(F,-d))$
$SomF(F)$	$\exists d(d > 0 \land Dist(F,d))$
$AlwP(F)$	$\neg SomP(\neg F)$
$AlwF(F)$	$\neg SomF(\neg F)$
$Lasted(F,t)$	$\forall d(0 < d < t \rightarrow Dist(F,-d))$
$Lasts(F,t)$	$\forall d(0 < d < t \rightarrow Dist(F,d))$
$WithinP(F,t)$	$\neg Lasted(\neg F,t)$
$WithinF(F,t)$	$\neg Lasts(\neg F,t)$
$Since(F,G)$	$\exists d(d > 0 \land Lasted(F,d) \land Dist(G,-d))$
$Until(F,G)$	$\exists d(d > 0 \land Lasts(F,d) \land Dist(G,d))$
$UpToNow(F)$	$\exists d(d > 0 \land Lasted(F,d))$; $Dist(F,-1)$ if $Time$ is discrete
$NowOn(F)$	$\exists d(d > 0 \land Lasts(F,d))$; $Dist(F,1)$ if $Time$ is discrete
$LastTime(F,t)$	$t \geq 0 \land Dist(F,-t) \land Lasted(\neg F,t)$
$NextTime(F,t)$	$t \geq 0 \land Dist(F,t) \land Lasts(\neg F,t)$
$Becomes(F)$	$UpToNow(\neg F) \land (F \lor NowOn(F))$

TABLE I

TRIO derived temporal operators

modeled as a state. A generic time-dependent predicate S represents a state if it obeys the following behavior (24):

$$\begin{aligned} & (UpToNow(S) \land S \land NowOn(S)) \\ \lor \quad & (UpToNow(\neg S) \land \neg S \land NowOn(\neg S)) \\ \lor \quad & (UpToNow(S) \land NowOn(\neg S)) \end{aligned}$$

$$\vee \qquad (UpToNow(\neg S) \wedge NowOn(S))$$

Every TRIO formula is considered implicitly temporally closed with a universal quantification, that is as if it was enclosed by a *Alw* operator, unless explicit existential quantification is provided, for example with a *Som* operator. Universal quantification over time expresses the fact that the formula is valid over the whole temporal axis.

TRIO formulae fall into one of three types: axioms, assumptions and theorems.

An *axiom* is a formula which is simply assumed to be valid. When describing a system in a specification, its basic behavior should be captured by a set of axioms relating the items it is composed of.

An *assumption* is a formula whose truth is postulated, but should be proved in a later phase of the development. In particular, we usually name *external assumption* a formula whose truth should be guaranteed by other parts of the specification composed with the present one. Conversely, an *internal assumption* is one that should be demonstrated by refinement, that is when details are added to the current specification. Note that the TRIO language does not distinguish between the two kinds of assumptions, which only pertain to metodological aspects, such as those considered in chapter 5.

Finally, a *theorem* is a formula whose validity can be proved formally through demonstration from other formulae. For sake of convenience, we will often use the word *lemma* to mean a theorem expressing an intermediate property, that is one which is not meaningful *per se* but only encapsulates a significant step to obtain a broader result. However, TRIO makes no distinction between theorems and lemmas, in that it only contemplates the former ones.

One last important feature of the TRIO language is that it is parametric with respect to the time model to be adopted, usually named *Time*. Hence, we can choose, between discrete and dense time models, what is best suited for our specification. In the remainder of this work, we will refer to a continuous time model, unless otherwise explicitly stated.

3.1.2 PVS encoding of TRIO in-the-small

An encoding of TRIO in-the-small in PVS (48), (49), (50), (51) constitutes what is usually called TVS (TRIO Verification System) or TRIO/PVS. More precisely, this tool consists of two parts: a set of theories containing the definitions of TRIO items and operators translated into the higher-order logic of PVS, and a set of proof strategies to automate and simplify the conduction of proofs with TRIO formulae in PVS. The TVS is extensively described in (24), (20). In this section we give the basic ideas about the encoding needed to understand the following chapters of this work.

TVS focuses mainly on TRIO specifications where the time domain is continuous, that is where $Time = \mathbb{R}$. The basic items are time-dependent terms (that is TD values) and formulae (that is TD propositions): they are simply defined as functions mapping *Time* onto a generic codomain D and onto Booleans, respectively.

```
TD_Term: TYPE = [Time -> D]
TD_Fmla: TYPE+ = TD_Term[bool]
```

Unlike TRIO formulae, where the current time instant is implicit, in TVS it is explicit so that every term is evaluated with respect to a given time instant. For example, consider the definition of the *Dist* operator in PVS:

```
Dist(T, t)(ct): D = T(ct+t)
```

where `T` is a time-dependent term whose codomain is `D` and `ct` stands for current time.

Of course, we need to extend the definitions of common Boolean connectives and of quantificators to handle time-dependent terms. For example, consider the AND operator representing conjunction; this is extended to time-dependent formulae as:

```
AND (A, B: TD_Fmla)(ct): bool = A(ct) AND B(ct)
```

Note that the result is still a function of time `ct`, that is a time-dependent formula. The definition means that $A \wedge B$ is true at time instant ct if and only if both A and B are true at ct.

Consider also the translation, under the name `FA`, of the universal quantification for time-dependent terms:

```
FA (A: [D -> TD_Fmla]): TD_Fmla =
   LAMBDA(ct): FORALL(x: D) : A(x)(ct)
```

The definition means that $\forall x A(x)$ is true at time instant ct if and only if $A(x)$ is true at ct for all possible values of the free variable x.

With these basic encodings, all the derived TRIO operators are translated naturally from their definitions. As an example, consider the translation of the operator $Lasts_{ee}$:

```
Lasts_ee(A: TD_Fmla, t: nnt)(ct): bool =
        FORALL (d : {d: Time | 0 < d AND d < t}) : A(ct+d)
```

where `nnt` is non-negative time (i.e. $\mathbb{R}^+$).

While in TRIO every formula is implicitly temporally closed with a universal quantification over time, in TVS it is not so, and the user must provide explicit quantification with the Alw operator. On the other hand, free variables are implicitly universally quantified by PVS, unless the user adopts explicitly another kind of quantification.

Obviously, TRIO axioms and theorems can be translated naturally into PVS axioms and theorems (and lemmas). On the other hand, the PVS keyword `ASSUMPTION` has a peculiar semantics which does not reflect the use TRIO does of the same keyword. TRIO assumptions may be translated using the PVS keyword `CONJECTURE`, which is a synonim for theorem, in the PVS language. More will be said about using TRIO assumptions in PVS in chapter 5.

Describing TVS proof strategies in a certain detail is out of the scope of this work. As a general idea, these strategies basically achieve two different goals.

First, a set of so called *pretty-printing* strategies serves to rewrite the proof sequent at each step in order to change its formulae to make them more TRIO-like, thus hiding the details of the encoding of TRIO in PVS. These strategies consist of a number of auto-rewrite rules and are applied automatically by the proof engine each time the sequent changes.

Second, other strategies enrich the PVS proof commands with a number of new (derived) commands, to make proofs of translated TRIO formulae easier to carry out in PVS. These commands take charge of ordinary formula manipulations, instantiations and rewritings to let the prover realize when two formulae are equivalent, to expand common definitions of TRIO operators, etc.

In chapter 6 we are going to design new proof strategies to handle the modular features of TRIO.

3.2 Modular features of TRIO

The TRIO language has a number of object-oriented constructs to support inheritance, genericity and modularization when specifying large and complex systems. The basic encapsulation unit is the *class*: a TRIO class is a collection of items, formulae and objects, that is instances of other classes. Note that TRIO does not have the notion of object construction and destruction, since it is a logic language. Objects encapsulated in classes are called *modules* of the class. Each class has an *interface*, defined as the set of items and formulae declared as *visible* in the signature section of the class. Defining an interface allows information hiding when writing modular specifications.

We usually distinguish between *simple* classes, that is classes without inner modules, and *structured* classes, that is classes built composing other classes together. Whether simple or structured, the semantics of a class is defined by the logical conjunction of all formulae of the class and of its modules. We can also define *arrays* of modules, containing multiple instances of the same class.

Classes can be generic with respect to a number of *parameters*; these parameters must be instantiated when the class is used as a module of another class. More precisely, a parameter can be any of values, domains or other classes.

An important feature, which is also typical of object-oriented languages, is *inheritance*. Each class can inherit from other classes, thus getting their items, formulae and modules. Inherited things can also be redefined or renamed, exploiting polimorphism, and multiple inheritance is allowed.

A special clause of the language is used to define *connections* between items of two modules: if two items are connected it means they are to be considered logically equivalent. Connections can be of three kinds: direct, cartesian and broadcast.

A *direct* connection connects items of the same arity: for example a time-dependent predicate I_1 to another time-dependent predicate I_2. This would be written in TRIO as `(direct I_1 I_2)`.

A *cartesian* connection connects each of the items of a $(1, \ldots, m)$ predicate I_1 to each of the items of another $(1, \ldots, n)$ predicate I_2, pairwise. This would be written in TRIO as `(cartesian I_1 I_2)` and its semantics is:

$$\forall i \in \{1, \ldots, m\} : \forall j \in \{1, \ldots, n\} : (I_1(i) = I_2(j))$$

Finally, a *broadcast* connection connects a simple time-dependent predicate I_1 to each of the items of a $(1, \ldots, n)$ predicate I_2. This would be written in TRIO as `(broadcast I_1 I_2)`, and its semantics is:

$$\forall i \in \{1, \ldots, n\} : (I_1 = I_2(i))$$

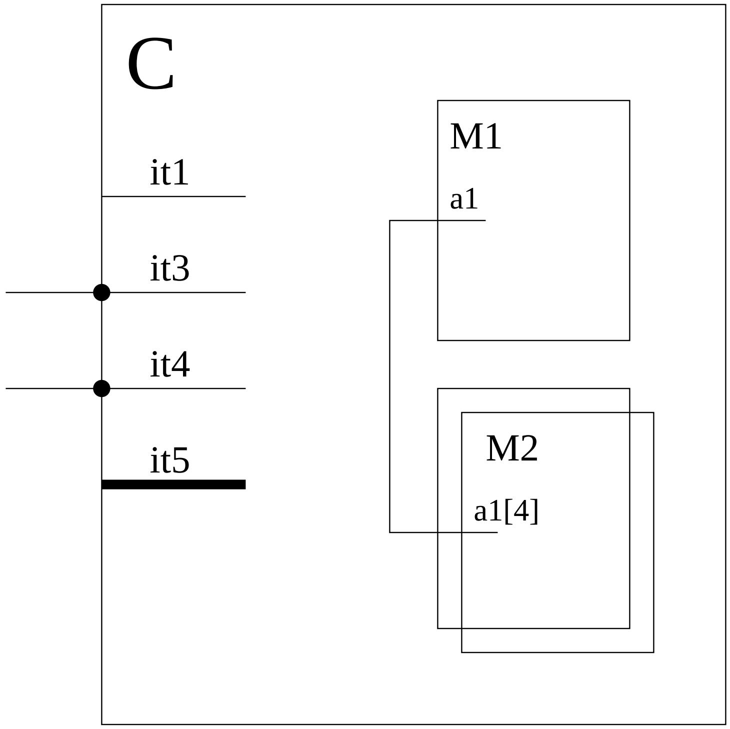

Figure 1. Graphical representation of a TRIO class

Note that TRIO semantics does not associate any direction to a given connection, even if most of the times one naturally thinks of a direction associated to a pair of connected items, as if information flowed from one end of the connection to the other.

Each class also has a graphical representation. This is often of help to visualize better a complex specification and to understand its structure. Figure 1 represents a sample structured class, whose TRIO specification is given below. This also serves as an example of the syntax of some of the modular features.

```
class C
  //class parameters
  ( const cst, domain DOM )

  //class C inherits from class B
  inherit: B [ redefine it1;
               rename it2 as it3 ]

  //classes to be used as modules must be imported
  import: A

  signature:

    //visible items and formulae of the class
    visible: it3, it4, th1;

    //temporal model is assumed continuous
    temporal domain: real;

    //items of the class
    items:
      TD it1 (arg1: DOM); //this is a redefinition
      event it4;
      state it5;

    //modules (i.e. class instances)
    modules:
      M1: A;
      M2: array [1..cst] of A; //array of modules

    //connections of the class
    connections:
      (direct M1.a1, M2[4].a1);

  //formulae of the class
  formulae:

  axiom ax1:
  it4 -> SomF(Lasts(it5, 3));

  theorem th1:
  Becomes(it3) -> WithinP(it4, 8);
```

```
end
```

Currently, there is no encoding of the modular features of TRIO in PVS. Chapter 4 will provide such an encoding, while chapter 6 will describe proof strategies to help the conduction of modular proofs in PVS with the given encoding.

CHAPTER 4

THE ENCODING OF TRIO IN PVS

As described in chapter 3, an encoding of TRIO in-the-small in PVS is currently available and used. In order to provide a verification system that allows the use of the modular features of the TRIO language, the first thing to do is to devise an encoding of these features into the PVS system, based on the current implementation of TRIO in-the-small. This chapter describes such an encoding.

More precisely, section 4.1 describes the mapping of classes, both basic and structured, thus showing how to translate the TRIO importing mechanisms. Section 4.2 considers the issue of visibility and namely to which extent it is possible to preserve in PVS the information hiding provided by the visibility clauses of TRIO classes. Finally, section 4.3 considers the inheritance mechanisms of TRIO and how they can be translated into PVS.

Before discussing these issues, a general consideration on the mapping is of order here. When describing the encodings of the modular features of TRIO, we should always think as if an automatic translation system from TRIO to PVS was available. Hence, the end user (that is she/he who writes a specification in TRIO) does not ideally have to know the details of the encodings but can concentrate only on the higher level description of the system.

4.1 TRIO classes

4.1.1 Basic issues

The basic encapsulation mechanism in TRIO is the *class*: a TRIO class is mapped onto a PVS *theory*. Each class is parametric with respect to an arbitrary number of parameters; each parameter can be a constant value, a domain or a class. This realizes the feature of *genericity* of the TRIO language.

PVS theories can have parameters as well. Hence, each TRIO parameter is naturally mapped onto a parameter in a PVS theory: constants are mapped onto constants and domains are mapped onto types. Unfortunately, PVS theories cannot be parametric with respect to other theories. There does not seem to be any solution to this limitation at the moment, so let us avoid this case for now.

Furthermore, each PVS theory that represents a TRIO class has an additional mandatory parameter named *instances* of a generic non empty type `TYPE+`. This parameter should be listed as the first in each PVS theory and it is used to translate in PVS the notion of TRIO module, as will be discussed shortly.

Each item in a TRIO class is obviously translated into a PVS item of the same type, using the encoding of TRIO in-the-small. However, each item should be made parametric with respect to a parameter of type `instances`. In other words, whenever we have a TRIO item `I` that whould be translated into the PVS item `I: T` of type `T`, we need instead to translate it to the item `I: [instances -> T]` of type `[instances -> T]`, i.e. function from `instances` to `T`. This allows the declaration of arrays of modules, as will be discussed shortly.

As an example, consider the translation of the following very simple TRIO class.

```
class simple

signature:

visible:
  A, B;

temporal domain: real;

items:
  event A;
  state B;
  TI total C(natural): integer;

formulae:
  axiom ax:
  A -> Lasts(B, 3);

  theorem th:
  B -> SomP(A);

end
```

It should be translated into the following PVS theory.

```
simple [instances: TYPE+]
   : THEORY

BEGIN

   IMPORTING trio_base, states_and_events

   A: [instances -> Event]
   B: [instances -> State]
   C: [instances -> [natural -> integer]]


   inst: VAR instances

   ax: AXIOM
```

```
   Alw( A(inst) IMPLIES Lasts(B(inst), 3) )

   th: THEOREM
   Alw( B(inst) IMPLIES SomP(A(inst)) )

END
```

As clearly shown in the example, since every item has been made parametric with respect to a variable of type `instances`, we need to make explicit that value whenever we reference any item in the theory.

4.1.2 Importing multiple instances

Let us now describe how the `instances` parameter should be used. The main problem we encounter in translating modules is that PVS cannot distinguish between different instances of the same theory, so that we have to simulate this mechanism somehow. In fact, we can use theories from within other theories by using the *importing* keyword. However, we cannot import multiple instances of the same theory, since PVS would consider them as indistinguishable, while TRIO considers different modules of the same class as logically distinct components. To solve this problem, we introduced the additional parameter `instances`. Whenever we import a theory as a module into another theory, we declare a new type of our choice. Then, we import the theory with that type as parameter (coupled with the `instances` type of the importing class, to allow the current class to become, in turn, module of another class). Hereinafter, whenever we reference to that module we give its full instantiation parameters, so that PVS can distinguish between different importings of the same module. For example, consider a class `structured` that has two modules of class `simple`. Its TRIO declaration is:

```
class structured

import: simple;

temporal domain: real;

modules:
  M1, M2: simple;

end
```

Its PVS translation would then be:

```
structured [instances: TYPE+]
   : THEORY

BEGIN

  M1_type: TYPE = {n: nat | n = 0} CONTAINING 0
  M2_type: TYPE = {n: nat | n = 0} CONTAINING 0

  IMPORTING simple[[instances, M1_type]], simple[[instances, M2_type]]

END
```

Even if the choice for the importing types is in general free, using an integer number is often a good choice since it is simple and works well with PVS (differently than, for example, enumeration types which are not very flexible in PVS). Note that the choice of the integer constant to represent the type is absolutely arbitrary, unless arrays are involved (they are discussed below). In particular, in the example we used the same integer twice, for two different modules, which is perfectly acceptable. After an importing, whenever we reference an item of the imported class we have to fully list the instantiation parameters, so that PVS can disambiguate between the two importings. For example a TRIO theorem of the form

```
theorem struc_th:
M1.A -> NowOn(M2.B)
```

should be translated into the PVS theorem

```
M1: VAR [instances, M1_type]
M2: VAR [instances, M2_type]

struc_th: THEOREM
Alw( A(M1) IMPLIES NowOn(B(M2)) )
```

In case we want to give explicit representation of the class the modules come from, we may verbosely write the same theorem as:

```
struc_th: THEOREM
Alw( simple[[instances, M1_type]].A(M1) IMPLIES
           NowOn(simple[[instances, M2_type]].B(M2)) )
```

PVS allows the user to give a particular name, or alias, to a theory whenever it is imported into another. This is done with the `AS` clause of the importings. It may seem at first that this may be used to better simulate the TRIO dotted notation to reference items of imported classes. Unfortunately, this does not work in general, since the `AS` alias is lost whenever the importing class is in turn imported into another one. Therefore, it should not be used in general. On the contrary, we will sometimes use it in the examples in the following chapters. The only reason to do that is to let the user read the PVS code more easily. For the same reason, we will sometimes import a class without instantiating it with the pair `[instances, module_type]`, using simply `module_type` when we are sure that we will not use the same class as module of another class. However, it is generally not advisable to do that and should be considered only as an explanatory aid usable whenever no other abiguities may arise.

Let us now consider the translation of array of modules from TRIO to PVS. This is where the parametrization of items with respect to a `instances` variable comes into play. Let us suppose we want to translate the following importing of a TRIO array of `simple` modules.

```
M3: array[2..4] of simple;
```

To do that, we declare an `instances` type corresponsing to the type of the index of the array. Then we use that as importing type for the class `simple`.

```
M3_type: TYPE = {n: nat | 2 <= n AND n <= 4} CONTAINING 2

IMPORTING simple[[instances, M3_type]];
```

Now, since every item in `simple` was parametric with respect to the importing type, we have as a result that each index in the range `2..4` refers to a logically distinct version of those items, one for each instance of the module in the array.

4.1.3 Connections

Another important feature of structured TRIO classes is the notion of *connection*. A connection is a logical equivalence between two items of two classes. A connection can be translated into a PVS equality (`=`). We chose to use the equality instead of the `IFF` operator, because equalities are automatically treated as rewrites in proofs, so that they are used more effectively by the automated prover, requiring less user interaction.

More precisely, we introduce a predicate *connect*() to indicate connections: it simply is a synonym for equality and is declared as a binary operator in the theory `TRIO_modular` that serves as a container for definitions used in PVS translations of modular features of TRIO.

Obviously, this service theory should be imported whenever we use those constructs in another theory.

```
H1, H2: VAR T  %T is a generic type

connect(H1, H2): boolean = (H1 = H2)
```

Connections should then be declared as axioms in PVS theories. We also give a standard naming for the connection axioms, to allow automatic translation of TRIO classes and to make the definition of proof strategies simpler (see chapter 6). If we have just one connection axiom listing all the connection equalities, linked by `AND`s, we call it `connections`. On the other hand, if we choose to split the connections among n axioms, we name them `connection_1`, `connection_2`, ..., `connection_n`. We note explicitly that all the TRIO connections can be translated with this mechanism, not only those of type `direct`. In fact, we just need to use correctly the instantiation variables we have seen in action above. For example, consider a *direct* connection between `M1.A` and `M2.A`. This would be written in PVS as:

```
connections: AXIOM
connect( A(M1), A(M2) )
```

A *cartesian* connection declared in TRIO as:

```
(cartesian M1.B M3.B);
```

would instead be translated in PVS with the aid of the instantiation variable `M3`.

```
M3: VAR [instances, M3_Type]

connect( B(M1), B(M3) )
```

This has the same semantics as in TRIO, since it corresponds to the formula:

$$\forall M_1 \in \{0\} : \forall M_3 \in \{2, 3, 4\} : B(M_1) = B(M_3)$$

4.2 The visibility issue

Up to now, we avoided discussing how to translate the notion of visibility of items and formulae from TRIO to PVS. We discuss this issue in this section, but we immediately have to say that the results will be rather negative.

The first idea that comes into mind is to use the `EXPORTING` clause of PVS to translate TRIO visibility. Whenver an item or a formula is visible we export it from the corresponding PVS theory. By doing this, the importing theories will only have access to those items and formulae declared as visible in the imported class.

Unfortunately, this mechanism does not work for two reasons. On the one hand, the exporting mechanism requires to export, together with the desired items and formulae, all the other items, types and theories referenced to by the exported objects. It is often difficult to correctly identify all the PVS dependencies so that we often have to use the clause `WITH CLOSURE` that automatically adds to the export list what is needed. This often results in exporting too many things, even those we do not want to be visible outside the current class. On the other hand, the exporting mechanism has a fundamental limitation: it does not handle nested theories in a correct way. In fact, consider a class A importing a class B. Then, class A cannot choose which items of B to export: it can either export them all or none of them. This is obviously an unacceptable limitation, since we often have the need to "propagate" the visibility of some items outside the current class, while hiding some others. Note that redeclaring in A all the items we want to export and connecting them with the corresponding items of B is not a solution, since the PVS system would then ask to export all the items in B as well, because of the dependency caused by the connection.

Another serious problem connected with the management of importings, exportings and visibility is the fact that PVS does not keep an internal representation of the nesting levels of the various theories we are using but simply considers them all at the same level, thus flattenting the representation which becomes very different from that in TRIO.

In consideration of these problems, we chose not to translate the visibility notion of TRIO classes at all. By doing this, we implicitly rely on the role of automatic translation tools in adhering to the semantics of the TRIO specification and avoiding references not allowed by information hiding. We believe that this solution, though not very satisfactory, is the best possible in consideration of the present limitation of PVS. It is possible that new versions of PVS will adopt better management of importing and exporting mechanisms, and that the TRIO mapping will be consequently enhanced. For now, an additional effort is required.

4.3 Class inheritance

A powerful feature of the TRIO language that permits the reuse of specification code is the *inheritance* mechanism. The PVS system guarantees the reuse of code by means of its importing mechanism, already discussed in translating TRIO modules in section 4.1. At a first glance, it may seem that the PVS importing mechanism can be effectively used to translate inheritance between TRIO classes, with even less problems than those encountered when mapping TRIO modules. Unfortunately, this is not the case. The PVS importing mechanism has a semantics that is very different from that of TRIO inheritance, so that we cannot provide a mapping which is both simple and effective. Let us see where the problems lie.

Let us first consider what happens when we only add new items into an inheriting class. In this case the importing mechanism seems to translate correctly the TRIO semantics. In fact, let us consider a basic class `A` with the following simple declaration.

```
class A
...
items
  TD it1;
...
end
```

Now consider another class `B` inheriting from `A` and adding another time-dependent item `it2`.

```
class B

inherit: A;
...
items:
  TD it2;
...
end
```

Class `A` would obviously be translated in PVS as:

```
A [instances: TYPE+]
  : THEORY
BEGIN
...
   it1: [instances -> TD_Fmla]
...
END A
```

On the other hand, `B` could be translated using the importing mechanism as:

```
B [instances: TYPE+]
  : THEORY
```

```
BEGIN
...
   IMPORTING A[instances]

   it2: [instances -> TD_Fmla]
...
END B
```

Now, consider what happens if a third class `C` has a module of class `B` and an item also named `it1`, and wants to reference the item `it1` of class `A` in one of its formulae.

```
class C
...
import: B;

signature:
   items: TD it1;
   modules: M: B;
...
   formulae:

   axiom ax1:
   M.it1 -> Past(it1, 4);

end
```

Here it is how to translate class `C` in PVS.

```
C [instances: TYPE+]
  : THEORY
BEGIN

  M_type: TYPE = {n: nat | n = 0} CONTAINING 0

  IMPORTING B[[instances, M_type]]

  it1: [instances -> TD_Fmla]

  inst: VAR instances
  M: VAR [instances, M_type]
```

```
  ax1: AXIOM
  Alw( it1(M) IMPLIES Past(it1(inst), 4) )

END C
```

As you can see, everything seems to work fine. However, `it1` is still considered by PVS an item of class `A`, ignoring the fact we are importing it from `B` instead. This is due to the "flattening" of import chains done by PVS, so that is does not contemplate an item `B.it1`. Obviously, this behavior is in general not acceptable when translating a TRIO class, since the user instantiating class `B` does not have to know which items come from class `A` and which have been declared directly in `B`. In most cases, however, this discrepancy can be safely ignored, since ambiguities can be solved by using the instantiation parameters.

However, other problems arise when trying to translate inheriting classes that also redefine items or formulae. The basic fact is that PVS does not have a real renaming mechanism, but only allows overloading. Hence, if we redefine an item from a inherited class, we do not really replace it with the new definition. On the contrary, the inherited item is still there, and may provoque ambiguities and indesired behaviors. Furthermore, whenever we redefine an item, that is we give a new definition under the same name, all the formulae declared in the class we are inheriting from are not considered applied automatically to the newly defined item, but still refer to the previous one, which is still there. An example will show more clearly this problem. Consider a TRIO class `X` that declares an item `x1` and a formula predicating about that item.

```
class X
...
  items: TD x1;
...
```

```
  formulae:
    axiom x_ax1:
    Som(x1);
...
end
```

Now consider another TRIO class `Y` that inherits from `X` and redefines its item `x1`.

```
class Y

inherit: X [redefine x1];
...
  items: event x1;
...
end
```

The TRIO semantics for inheritance tells that the axiom `x_ax1` refers to the new declaration of item `x1` in class `Y`, as an effect of the redefinition. On the other hand, the PVS translation of class `Y` would be, using the importing mechanism:

```
Y [instances: TYPE+]
  : THEORY
BEGIN

  IMPORTING X[instances]

  x1: [instances -> Event]
...
END Y
```

Unfortunately, PVS does not refer axiom `x_ax1` to the *event* `x1`, but still refers it to *time-dependent formula* `x1` declared in theory `X`. More precisely, this latter item is still available under the name `X.x1`. This is because formulae always refer to the "nearest" items, that is those declared in the same theory as the formula is. In particular, the problem would not be solved by moving the importing *after* the new declaration of `x1` since the axiom would still refer to `X.x1`.

All in all, it seems that there is no safe way to exploit the importing mechanism of PVS to map TRIO inheritance mechanisms. Hence, we still need to rely on (still to be developed) automated translation tools to correctly map this important TRIO feature. More precisely, the translator should rewrite the class we are inheriting from, adding the things that need to be added, redefining the things that need redefining and keeping the changes consistent with the TRIO specification. As an aside, note that this guideline also works when dealing with multiple inheritance, since the needed renamings can also be managed during the translation.

A final example will now show a case of multiple inheritance and how it should be translated in PVS. We are specifying a *flip-flop* logic device. We fist of all describe it very generically as an object that has a Boolean state.

```
class flip_flop

signature

  visible: Q;

  items:
     state Q;
     TI total tau: real;

end
```

The class has no axioms. We first redefine it by adding a *set* command.

```
class set_flip_flop
```

```
inherit: flip_flop;

signature

  visible: S;

  items:
     event S;

formulae:

  axiom set:
  S -> Futr(Q, tau);

end
```

Another, different refinement of `flip_flop` is obtained by adding a *reset* command.

```
class reset_flip_flop

inherit: flip_flop;

signature

  visible: R;

  items:
     event R;

formulae:

  axiom set:
  R -> Futr(not Q, tau);

end
```

Now we define a *set-reset* flip-flop as a device with both a set and a reset command and whose state stays unchanged if no command is issued.

```
class set_reset_flip_flop
```

```
inherit: set_flip_flop, reset_flip_flop;

formulae:

  axiom persistency:
  (not S & not R) ->
     ((Q -> Lasts(Q, tau)) & (not Q -> Lasts(not Q, tau)));
end
```

Finally, a *J-K* flip-flop is a set-reset flip-flop where the state is complemented if both `S` and `R` (now conventionally named `J` and `K`) are on at the same time.

```
class jk_flip_flop

inherit: set_reset_flip_flop [redefine set, reset;
                              rename S as J;
                              rename R as K];

formulae:

  axiom set:
  (J & not K) -> Futr(Q, tau);

  axiom reset:
  (K & not J) -> Futr(not Q, tau);

  axiom commutation:
  (J & K) -> (Q <-> Futr(not Q, tau));

end
```

The above classes would be translated in PVS as follows.

```
flip_flop [instances: TYPE+]
  : THEORY
BEGIN

  Q: [instances -> State]
  tau: [instances -> real]

END flip_flop
```

```
set_flip_flop [instances: TYPE+]
  : THEORY
BEGIN

  Q: [instances -> State]
  tau: [instances -> real]
  S: [instances -> Event]

  inst: VAR instances

  set: AXIOM
  Alw( S(inst) IMPLIES Futr(Q(inst), tau) )

END set_flip_flop

reset_flip_flop [instances: TYPE+]
  : THEORY
BEGIN

  Q: [instances -> State]
  tau: [instances -> real]
  R: [instances -> Event]

  inst: VAR instances

  reset: AXIOM
  Alw( R(inst) IMPLIES Futr(NOT Q(inst), tau) )

END reset_flip_flop

set_reset_flip_flop [instances: TYPE+]
  : THEORY
BEGIN

  Q: [instances -> State]
  tau: [instances -> real]
  S: [instances -> Event]
  R: [instances -> Event]

  inst: VAR instances

  set: AXIOM
```

```
  Alw( S(inst) IMPLIES Futr(Q(inst), tau) )

  reset: AXIOM
  Alw( R(inst) IMPLIES Futr(NOT Q(inst), tau) )

  persistency: AXIOM
  Alw( (NOT S(inst) AND NOT R(inst)) IMPLIES
      ((Q(inst) IMPLIES Lasts(Q(inst), tau))
       AND (NOT Q(inst) IMPLIES Lasts(NOT Q(inst), tau))) )

END set_reset_flip_flop

jk_flip_flop [instances: TYPE+]
  : THEORY
BEGIN

  Q: [instances -> State]
  tau: [instances -> real]
  J: [instances -> Event]
  K: [instances -> Event]

  inst: VAR instances

  set: AXIOM
  Alw( J(inst) IMPLIES Futr(Q(inst), tau) )

  reset: AXIOM
  Alw( K(inst) IMPLIES Futr(NOT Q(inst), tau) )

  persistency: AXIOM
  Alw( (NOT J(inst) AND NOT K(inst)) IMPLIES
      ((Q(inst) IMPLIES Lasts(Q(inst), tau))
       AND (NOT Q(inst) IMPLIES Lasts(NOT Q(inst), tau))) )

  commutation: AXIOM
  Alw( (J(inst) AND K(inst))
       IMPLIES (Q(inst) IFF Futr(NOT Q(inst), tau)) )

END jk_flip_flop
```

CHAPTER 5

A COMPOSITIONALITY FRAMEWORK WITH TRIO

When specifying the behavior of a component or of an open system, one often needs to make some form of assumption about the behavior of the *environment* interacting with the component, that is the outside part of the world that provides the inputs to the open system. In fact, it is often the case that a given component mantains certain expected properties only if its environment behaves in some constrained manner: if the inputs are instead completely unpredictable, it may be impossible to design a system that still behaves correctly. For example, the input channels may be communication channels where we expect that every user of the channel adheres to a certain communication protocol. In other cases, we may simply want to assume that a voltage signal is discrete, so that it takes only a number of known values and avoids all the other possible ones.

It is easy to realize that this kind of situation often happens in practice, so that we want to be able to specify and reason about such kind of systems. What we need to do is to include the constraints imposed on the environment into our formal model. Then, we can prove the properties of each module so that they are satisfied if we assume the environment constraints to be true. Finally, when we compose modules into a larger system, we want to guarantee that whenever a module M_1 provides input to another module M_2, then M_1 also satifies the environment constraints required by M_2, either directly or by demanding them to its own environment.

In order to achieve this expressiveness into a formal specification, we can basically choose between two approaches: the *rely/guarantee* approach or the *lazy* compositional approach. See sections 2.1.2.1 and 2.1.2.2 respectively for a review of the recent literature about these topics. It is important to point out that these two paradigms must not be considered incompatible. In fact, it is perfectly possible to mix them into the specification of the same system, using the one more suited in each part.

In the following sections, we consider the rely/guarantee compositional approach with reference to the language TRIO and point out some aspects one must take into account when doing a compositional proof of a system specified as such with TVS.

5.1 A rely/guarantee specification

Let us consider a module specified as a TRIO class C. The *environment* of C is everything outside C that constrains in some manner its visible input items, or a part of them.

In TRIO, there is no pre-defined notion of input, output or shared items; even when we define a connection between the items of two modules, there is no semantic notion of direction associated to it, so that it goes from an output to an input. However, it is often the case that one has such a kind of distinction in mind when specifying a system and partitioning it into submodules. In general, we can say that even if rely/guarantee reasoning usually has some notion of input/output variables, we can simply assume that an environment assumption is a property characterizing the behavior of a number of visible items of the class.

In TRIO, the environment assumptions of a class can be expressed with the keyword `assumption`. Let us suppose that the class C has an assumption E about its environment. Therefore, the declaration of the class is:

```
class C

  signature:

    visible:
    i1, i2, ..., in;


  formulae:

    assumption E:
    P_e(i1, i2, ..., in);
```

We have explicitly shown that the assumption E is a predicate over the visible items $i_1, \ldots, i_n$ of the class only. This is a well-defined restriction on the semantics of the keyword `assumption` of the TRIO language. More precisely, we will only refer to what is usually called an *external* assumption, that is one about visible items only.

An assumption is used with reference to one or many derived properties of the system, which in TRIO can be expressed with the keyword `theorem`. We want to explicitly link a number of assumptions to a theorem that relies on them to be proven. The TRIO language does not have an *ad hoc* feature to explicitly show this. For sake of brevity, we adopt a new non-standard keyword `rely on` as an optional part of every theorem. This will just be a shorthand for longer TRIO formulae, as it is explained below, and is not meant to be considered a feature of the TRIO language, but just an explanatory aid.

More precisely, we have the following general scenario in mind when writing rely/guarantee specifications in TRIO. One specifies the basic behavior of a class in terms of axioms. The axioms are usually formulae over both visible and non visible items, and usually rely on no assumptions about these items, since they just state the basic behavior of the class. In order to characterize the class externally, one usually derives a number of remarkable properties as theorems of the class. In order to prove them, she/he must usually make some additional assumptions about the behavior of the environment (i.e. the visible items), expressed as `assumptions`. Among the deducted properties, those predicating about visible items only constitute the *interface abstraction* of the class and characterize its external behavior under the given assumptions. Of course, we can give different environment assumptions of different strengths, if more than one possible environment scenario is possible. Usually, the stronger the environment assumptions a theorem relies on, the stronger the property it can express is.

The declaration for a theorem M of the aforementioned class C would be written as:

```
formulae:

  ...

  theorem M:
    rely on: E;
  P_m(...);
```

We need to understand the precise semantics of a rely/guarantee property, so that we are able to carry out formal reasoning about it and we can implement this TRIO formula into PVS to build automated proofs. One first obvious meaning of a rely/guarantee formula is logical implication: property M relies on the environment assumption E, meaning that the formula

$E \Rightarrow M$ is a valid property of the class C. Therefore, the above rely/guarantee theorem M could be translated into plain TRIO as:

```
theorem M:
Alw(P_e(...)) -> Alw(P_m(...))
```

and in PVS as follows.

```
M: THEOREM
Alw(P_e(...)) => Alw(P_m(...))
```

We have chosen to use the operator `=>` instead of the keyword `IMPLIES` in order to distinguish between the operator used in TRIO formulae with time-dependent items as arguments and the rely/guarantee implication which relates temporally closed TRIO formulae that do not refer to a single instant of time (i.e. quantified with a $Alw()$ or $Som()$ operator). Of course, whenever a theorem relies on more than one assumption, this means that the conjunction of the assumption formulae implies the formula of the theorem.

Let us now see a simple complete specification of a class using a rely/guarantee scheme. The module has just a Boolean input and a Boolean output. Figure 2 shows the very simple interface of the module.

We adopt a discrete temporal model for this device because the following results become simpler and more understandable. However, the same can be done with the usual dense temporal model, just with some more details to be put in place. The basic behavior of the module is the following: whatever it receives on its input (`true` or `false`), it outputs it one time step later on its output. Moreover, at the beginning (time `0`) the module is initialized so that it outputs a `true` value, regardless of its input. This simple behavior is specified as follows:

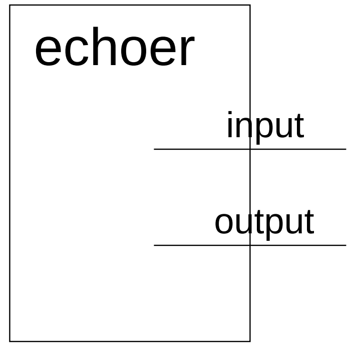

Figure 2. Interface of the `echoer` class

```
class echoer

   signature:
      visible:
         input, output;

      temporal domain: natural;

      items:
         TD input;
         TD output;

   formulae:

      axiom init:
      output(0);

      axiom in_to_out:
      (input -> Futr(output, 1)) &
        (not input -> Futr(not (output), 1));

end
```

Note that axiom `in_to_out` can be equivalently written as:

$(input \Rightarrow NowOn(output)) \wedge (\neg input \Rightarrow NowOn(\neg output))$, a consequence of the the temporal domain being discrete.

Now, we give a derived characterization of the behavior of the class with a rely/guarantee property. The stated behavior is the following: assuming the environment always inputs `true`, the component guarantees that its output is always `true`. This behavior is formally written in TRIO, enriched with the `rely on` notation, as:

```
assumption on_input:
input;

theorem rely_guarantee:
   rely on: on_input;
output;
```

where `on_input` is the assumption about the environment and `rely_guarantee` is the property linked with the environment behavior.

Now that the specification of this simple class is complete, we can build the proof for the (only) local derived property, that is the theorem `rely_guarantee`.

The proof is straightforward using the axioms and assuming `on_input` to be true and is easily carried out in a handful of steps with TVS. We report here a short summary of the proof, that can be safely skipped without compromising the understanding of the following sections. The goal to be proven is the temporally-closed TRIO formula:

$$Alw(input) \Rightarrow Alw(output)$$

Let t be a generic time instant. We distinguish two cases, whether t is equal to 0 or is not (that is it is greater than 0, the temporal domain being the natural numbers). The first branch of the proof requires to prove:

$$output(0)$$

which can be done by the axiom `init`. The other branch of the proof is represented by the formula

$$Alw(input) \Rightarrow (\forall t > 0 : (output(t)))$$

and is closed by using the axiom `in_to_out` and applying it at time $t - 1$.

5.2 Compositional inference rules for rely/guarantee systems

Now, we want to consider compositional reasoning with classes specified with the rely/guarantee paradigm. Whenever we compose a class C_1 with another class C_2 so that we connect some items of the two classes together, we want to be able to discharge the assumptions C_1 makes about its environment by means of some of the exhibited properties of C_2, or, by transitivity, by means of the environment assumptions of C_2.

Let us define this idea more precisely, with reference to a system composed of n modules $C_1, \ldots, C_n$. Each module C_i, $i = 1, \ldots, n$ has an interface abstraction specified synthetically as $E_i \Rightarrow M_i$. This means that its externally visible behavior is characterized by a rely/guarantee property of the form: if E_i is true of C_i's environment, then C_i exhibits the property

M_i. Needless to say, E_i and M_i can be arbitrarily complex TRIO formulae. Therefore, the composition of the n modules can be characterized by the formula:

$$\bigwedge_{i=1,\ldots,n} (E_i \Rightarrow M_i)$$

where we do not show explicitly the connections of items (which can be considered simply as renamings).

The composite class C is the composition of the n simpler classes we have just mentioned. In general, C also has its own environment and we can define an assumption on this environment and name it E. What we want to prove of C is that, assuming its environment behaves as in E, it guarantees a behavior M, that is:

$$E \Rightarrow M$$

Intuitively, one would expect the following circular inference rule to be valid.

Proposition 1 (Invalid rely/guarantee inference rule) *If, for* $i = 1, \ldots, n$ *the following two conditions hold:*

1. $E \wedge \bigwedge_{j=1,\ldots,n} M_j \Rightarrow E_i$
2. $\bigwedge_{j=1,\ldots,n} M_j \Rightarrow M$

then

$$Alw\left(\bigwedge_{j=1,\ldots,n} (E_j \Rightarrow M_j)\right) \Rightarrow Alw(E \Rightarrow M)$$

The intuitive meaning of Proposition 1 is simple: if the environment assumption of each module can be discharged by the global environment assumption E and the guarantees of the other modules (possibly including itself), and the global guarantee M can be inferred from the guarantees of the other modules, then the composition of the n modules has a rely/guarantee behavior $E \Rightarrow M$.

However, this intuitively reasonable and simple inference rule is not valid in general, since we have to make additional hypotheses about the assumptions of each module and also about how the rely of each module is linked to the guarantee of the same module. In order to show that, we make two similar examples where the hypotheses of proposition 1 hold, but nonetheless the conclusion is wrong in one of the two examples and true in the other one. These examples are modeled after those in (2), (4).

Let us consider a complete system built using two instances of the previously declared class `echoer`. This system simply connects the input of one echoer to the output of the other and vice-versa. It is shown in Figure 3, while here we list its TRIO specification:

```
class two_echoers

   import: echoer;

   signature:

      temporal domain: natural;

      modules:
        P1, P2: echoer;

      connections:
```

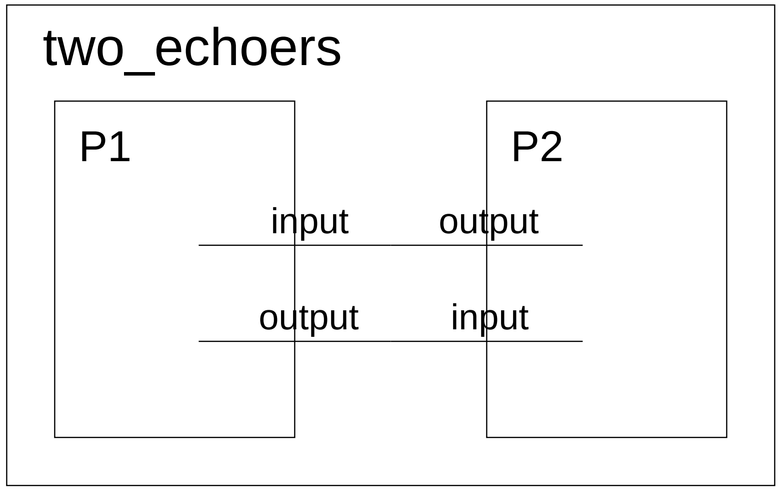

Figure 3. Interface of the `two_echoers` class

```
      (direct P1.input, P2.output);
      (direct P2.input, P1.output);

   formulae:

      theorem rely_guarantee:
      P1.output & P2.output;

end
```

The given system is a closed system, so that it does not have any global environment assumption. We apply the inference scheme of proposition 1 as follows: if

1. `P1.rely_guarantee` $\Rightarrow$ `P2.on_input`

2. `P2.rely_guarantee` $\Rightarrow$ `P1.on_input`

3. `P1.rely_guarantee` $\wedge$ `P2.rely_guarantee` $\Rightarrow$ `rely_guarantee`

then we conclude `rely_guarantee` is true, having already proved the rely/guarantee formulae `P1.rely_guarantee` and `P2.rely_guarantee` to be valid locally. And in fact the above conclusion is true and can be proved formally with TVS; some commented details of the proof are shown in section 5.6.

However, an apparently minimal modification in the definition of the class `echoer` suffices to make the whole reasoning incorrect. Let us consider this new modified class `echoer_2`.

```
class echoer_2

   signature:
      visible:
         input, output;

      temporal domain: natural;

      items:
         TD input;
         TD output;

   formulae:

      axiom init:
      output(0);

      axiom in_to_out:
      (input -> Futr(output, 1)) &
        (not input -> Futr(not (output), 1));

      assumption on_input:
      Som(not input);

      theorem rely_guarantee:
         rely on: on_input;
      Som(not output);

end
```

The only significant change is that we now adopt an existential closure of the formulae with respect to time, instead of the standard implicit universal closure. The consequently modified composite class `two_echoers_2` would only differ in the expected guaranteed property we want to prove:

```
class two_echoers_2

...

      theorem rely_guarantee:
      Som(not P1.output) & Som(not P2.output);

end
```

It is still very simple to prove that:

1. `P1.rely_guarantee` $\Rightarrow$ `P2.on_input`

2. `P2.rely_guarantee` $\Rightarrow$ `P1.on_input`

3. `P1.rely_guarantee` $\wedge$ `P2.rely_guarantee` $\Rightarrow$ `rely_guarantee`

and it also trivial to prove:

1. `Som(not P1.input)` $\Rightarrow$ `Som(not P1.output)`

2. `Som(not P2.input)` $\Rightarrow$ `Som(not P2.output)`

However, it is *not* true that `Som(not P1.output)` or that `Som(not P2.output)`. In fact, it is instead true that `Alw(P1.output & P2.output)`, since the axioms of the classes on which the previous proof was built have not changed, hence invalidating the inference rule in proposition 1.

In order to formulate a sound inference rule, we have to make additional assumptions about the validity of the properties used as assumptions of the classes. Moreover, we have to assume a stronger semantics for the rely/guarantee specifications of modules, i.e. stronger than simple implication. More precisely, we must link temporally the behavior of the environment and the one of the module, introducing a tight causal link between them.

Much of the work done about rely/guarantee compositional reasoning (see section 2.1.2.1), and especially (2), (4), bases its results on the safety characterization of the formulae used as assumptions and guarantees of the modules being composed. In order to see if something similar applies to our framework, section 5.3 investigates safety in TRIO. Unfortunately, the characterization will be shown to be of little practical use in formulating a sound inference rule. However, we can still get to a practically usable inference rule by considering additional hypotheses.

Under these respects, section 5.4 discusses how to strengthen a rely/guarantee specification, while section 5.5 below finally formulates a valid inference rule, based on the results of the previous sections.

5.3 Safety properties

An interesting classification of the temporal properties of a system is the one between *safety* properties and non-safety properties. A safety property (7), (57) is one that is finitely refutable, that is it can be proven false by observing a single violation at a single finite instant of time. The term "safety" intuitively indicates that we have such a class of properties whenever we specify the "safe" behavior of a system. In fact, by safe we usually mean a behavior where no bad thing ever happens. As it is clear from this informal definition, it is easy to disprove a safety property, since it suffices to find a single finite instant of time where the property does not hold. Most of the literature about rely/guarantee compositional reasoning (see section 2.1.2.1) bases its results on the safety characterization of the formulae used as assumptions and guarantees. This section tries to understand if this can be extended to the TRIO framework. We warn the reader that the results will be negative, in that the safety of TRIO formulae cannot be given a completely syntactical characterization.

First of all, let us precisely define what is a safety property. The basic idea is that we observe the validity of the formula over a finite time interval; if this suffices to falsify the formula, we have a safety property. In order to give a formal characterization, we use the notion of history: a *history* is any possible evolution of all the items of interest (i.e. those referred to in the formula we are considering) over the whole temporal axis: in other words, it is an interpretation for the given formula, or a candidate model of it. If h is a history and $I \subseteq Time$ is a time interval, by the expression $h[I]$ we denote the subset of the facts described by the history h over the time interval I only. A *completion* of a history h with respect to a non-empty time interval $I \subseteq Time$ is any other complete history h' such that: $h[I] = h'[I]$, and we write $h' \sim_I h$ (h' is a completion of h with respect to time interval I). This means that h' is the same as h over the time interval I, while can be anything anywhere else on the temporal axis. As a side remark, we note that the binary relation $\sim_I$ is an equivalence relation, for any given time interval I. With this in mind, we give the following definition.

Definition 1 (Safety) *Let T be a TRIO formula.*

T *is a* safety property *if and only if for every history h:*

$$h \not\models T \quad \Rightarrow \quad \text{exists a finite nonempty } I \subseteq Time : (\forall h' : (h' \sim_I h) \Rightarrow (h' \not\models T))$$

That is, if T is false we can prove it so by considering its behavior only over a finite time interval.

Note that the time interval I can be, in general, any finite union of disjoint finite time intervals; however, most of the practically interesting cases involve only one contiguos interval or even simply a point on the temporal axis.

We now want to give some practical characterizations of safety TRIO formulae, to see if this can be of use in formulating a rely/guarantee inference rule. Unfortunately, we cannot give a condition for safety which is both necessary and sufficient, and purely syntactical, as we will show below. Therefore, we just give one characterization of safety which is purely syntactical but is just a sufficient condition (i.e. there are formulae which do not satisfy the criterion but are nonetheless safety formulae).

5.3.1 Conditions for safety preservation

Let P be a primitive time-dependent formula, that is one defined without using any logical or temporal TRIO operator. Then P is by definition a safety property, since if it is false it is so at some instant. Every time-independent formula is also by definition a safety property, since safety is a temporal property and time-independent items do not affect it.

Let now T be a TRIO formula representing a safety property. If we use T as an argument of a TRIO operator, the resulting formula is still a safety property if the operator is one of those listed in Table II. In fact, for each of them we can decide if the formula is true or false for a given history by just considering what happens to the argument of the operator over a finite time interval. Since the argument is in turn a safety formula, this implies that its truth can also be decided on the basis of a finite time interval, so that the overall analysis is finite.

Safety formula	Finite interval(s) sufficient to determine the truth value
$Dist(T,t)$	$[t,t]$
$Past(T,t)$	$[-t,-t]$
$Futr(T,t)$	$[t,t]$
$Alw(T)$	$[t,t]$ for any time t
$AlwP(T)$	$[-t,-t]$ for any positive time t
$AlwF(T)$	$[t,t]$ for any positive time t
$Lasted(T,t)$	$[-t',-t']$ for any positive time $t'<t$
$Lasts(T,t)$	$[t',t']$ for any positive time $t'<t$
$Within(T,t)$	$[-t,t]$
$WithinP(T,t)$	$[-t,0]$
$WithinF(T,t)$	$[0,t]$
$LastTime(T,t)$	$[-t,-t]$ for T or $[-t,0]$ for $\neg T$
$NextTime(T,t)$	$[t,t]$ for T or $[0,t]$ for $\neg T$
$UpToNow(T)$	$[-t,0]$ for any positive time t
$NowOn(T)$	$[0,t]$ for any positive time t
$Becomes(T)$	$[-t,0]$ or $[0,t]$ for any positive time t

TABLE II

Safety-preserving TRIO temporal operators

Therefore, we say that the TRIO operators *Dist*, *Past*, *Futr*, *Alw*, *AlwP*, *AlwF*, *Lasted*, *Lasts*, *Within*, *WithinP*, *WithinF*, *LastTime*, *NextTime*, *UpToNow*, *NowOn* and *Becomes* preserve safety.

On the other hand, the other TRIO operators do *not* preserve safety. In fact, if T, T_1, T_2 are TRIO formulae expressing safety properties, the derived formulae listed in Table III are non-safety properties. For each of them, we may have to consider an infinite time interval to prove the property false, when nothing can be said for sure by observing the behavior over a finite time interval only.

NON-safety formula	Infinite interval(s) needed to determine the truth value
$Som(T)$	$[-\infty, +\infty]$
$SomP(T)$	$[-\infty, 0]$
$SomF(T)$	$[0, +\infty]$
$Until(T_1, T_2)$	$[0, +\infty]$ for T_1 and $\neg T_2$
$Since(T_1, T_2)$	$[-\infty, 0]$ for T_1 and $\neg T_2$

TABLE III

Non safety-preserving TRIO temporal operators

Following what has just been explained, it may seem that to decide the safety of a TRIO formula we can simply consider the temporal operators used to build the formula, on a purely syntactical basis: if they all are safety-preserving operators, then the formula is a safety formula.

Unfortunately, this is not true in general. In fact, we now pass to analyze what happens to the safety of a formula when we use the Boolean connectives $\neg$, $\wedge$ and $\vee$ and when we use explicit quantification over free variables. The result will be that we cannot formulate a purely syntactical condition for safety that is both necessary and sufficient.

In fact, let T_1 and T_2 be two safety formulae. Then the formulae $T_1 \wedge T_2$ and $T_1 \vee T_2$ are both still safety formulae. In fact, if the AND formula is false, then there is by definition some instant at which either T_1 or T_2 is false; this is recognizable since T_1 and T_2 are safety formulae and their truth can be decided analyzing finite intervals only. Similarly, if the OR formula is false, then there is some instant at which both T_1 and T_2 are false; this is also recognizable since it suffices to take the intersection of the intervals where T_1 and T_2 are independently false.

Let us now consider what happens to the safety of a formula when we apply quantification. It is trivial to note that, whenever the quantification is done on a non-temporal variable, safety is simply preserved. In fact, safety concerns the temporal properties of a formula, which are unchanged by a quantification over other domains. On the other hand, rules for safety preservation when quantification is done on temporal variables are hard to formulate. To see this, consider the following two quantified formulae.

$$\exists t : (0 < t \wedge t < 10 \wedge Dist(T, t)) \tag{5.1}$$

$$\exists t : (0 < t \wedge -1 < t \wedge Dist(T, t)) \tag{5.2}$$

The two formulae are syntactically the same, in that they are both existential quantifications of a temporal variable which is used in two time-independent items and in one time-dependent item (i.e. $Dist(T, t)$). However, Equation 5.1 is equivalent to the TRIO formula $WithinF(T, 10)$ and is a safety formula. On the other hand, Equation 5.2 is equivalent to $SomF(T)$ which is not a safety formula.

In order to distinguish these cases one needs to abandon a purely syntactical characterization and analyze the temporal intervals over which the quantified variables are allowed to vary. In fact, let us consider the subdomain for the quantified variable t which is mapped to TRUE values in the time-independent terms. This can be expressed as the set $D \subseteq Time : t \in D \Rightarrow f(t)$ where $f()$ is the Boolean functions we are considering. Another way to say the same thing is that D is the set defined by the Boolean formula $f(t)$ (i.e. $0 < t < 10$ and $t > 0$ respectively). Then, D is the bounded set $[0, 10]$ in Equation 5.1 and is instead the unbounded set $[0, +\infty]$ in Equation 5.2. So, in the first case safety is preserved because we can observe the term T over a finite interval only to determine its truth value, while in the second case we need to analyze the whole positive semiaxis.

Note that this particular behavior does not happen when we apply universal quantification. In fact, the formulae

$$\forall t : \neg((0 < t \wedge t < 10) \wedge \neg Dist(T, t)) \tag{5.3}$$

$$\forall t : Dist(T, t) \tag{5.4}$$

are both safety formulae: the first one is just a counterintuitive way of writing $Lasts(T, 10)$ and the second corresponds to $Alw(T)$.

However, the presence of an unbounded interval within an existential quantification does not always mean the loss of safety of a formula. Consider for example the definition of the $NowOn()$ operator in dense domains.

$$NowOn(T) \quad \triangleq \quad \exists t(t > 0 \wedge Lasts(T, t))$$

Even if the quantified variable t is allowed to vary in $[0, +\infty]$, nonetheless the formula expresses a safety property. This is because the second argument of the $Lasts()$ operator can be rewritten as part of a time-independent term. To see this better, we can expand the $Lasts()$ operator in the $NowOn()$ definition.

$$NowOn(T) \quad \triangleq \quad \exists t(t > 0 \wedge (\forall t'(0 < t' < t \Rightarrow Dist(T, t'))))$$

So the variable appearing as an argument to the $Dist()$ operator is t' which is bounded by t.

The effect of the Boolean unary operator $\neg$ on the safety of formulae can similarly be understood in connection with the behavior of the existential quantification. In fact, consider the two formulae

$$AlwF(T) \tag{5.5}$$

$$Lasts(T, 10) \tag{5.6}$$

which are both safety formulae.

The negation of the first one produces $\neg AlwF(T) \equiv SomF(\neg T)$ which is no more a safety property. On the other hand, the negation of the other formula is $\neg Lasts(T, 10) \equiv WithinF(\neg T, 10)$ which is instead still a safety property. If we rewrite the two formulae in terms of the only $Dist()$ operator and bring the negation inside the quantification, we have the following.

$$\neg(\forall t(t \leq 0 \vee Dist(T, t))) \equiv (\exists t(t > 0 \wedge \neg Dist(T, t))) \tag{5.7}$$

$$\neg(\forall t((t \leq 0 \vee t \geq 10) \vee Dist(T, t))) \equiv (\exists t(0 < t \wedge t < 10 \wedge \neg Dist(T, t))) \tag{5.8}$$

So in Equation 5.7 we have an existential quantification over an unbounded domain and we lose safety; in Equation 5.8 the domain is bounded and safety stays. In other words, what the negation has done in these two examples is to complement temporal intervals; when this complementing has produced an unbounded interval inside the domain of an existential quantification, safety has been lost.

5.3.2 A sufficient syntactical condition for safety

As seen in the previous section, finding out whether a TRIO formula expresses a safety property or not is far from trivial in the general case. However, if we restrict ourselves to a proper subset of all the TRIO formulae we can get to a sufficient condition for the safety of a formula which is purely syntactical. This condition may be too restrictive in the general case, but is surely of use in many practical situations.

The basic considerations about safety preserving and non-safety preserving TRIO operators done in the previous section can be applied. We avoid explicit quantification on temporal variables (while we allow it on variables other than time) and only use safety preserving operators on basic time-dependent items. We allow negation only if directly on basic time-dependent terms. To summarize, we have the following sufficient condition for safety.

Proposition 2 (Sufficient syntactical condition for safety) *Let F be any TRIO formula. If F is built from basic time-dependent and time-independent items by observing the following rules.*

1. *There is no use of explicit quantification over time variables*
2. *There is no use of TRIO operators other than the safety-preserving ones, listed in Table II*
3. *There is use of the negation operator directly on basic time-dependent and time-independent items only*
4. *There is use of the logical connectives $\wedge$ and $\vee$ only*

Then F is a safety formula.

This proposition can be proved inductively starting from basic time-dependent and time-independent items.

As an example, if we consider the formula $Alw(Lasts(WithinP(T_1 \vee T_2, t)))$ this is a safety formula, since $Alw()$, $Lasts()$ and $WithinP()$ are all safety-preserving operators. Conversely, consider the formula $Alw(Lasts(Until(T_1, T_2)))$. This is clearly a non-safety formula, since proving it false cannot be done by considering its behavior on finite intervals only, because of the $Until()$ operator.

5.3.3 Safety is of little use in a TRIO rely/guarantee framework

Definition 1 above gives a characterization of safety properties based on what is needed to falsify them: if we can prove the property is false by observing its behavior over a finite time interval, then it is a safety property. If we reverse the implication used in definition 1, we can get to another equivalent characterization of a safety property: if the property is true on every finite time interval (arbitrarily completed) then it is true on the whole temporal axis (otherwise we could falsify it). This is shown in the following definition.

Definition 2 (Safety - second definition) *Let T be a TRIO formula.*

T is a safety property *if and only if for every history h:*

$$\text{(for all finite nonempty } I \subseteq Time : (\exists h' : (h' \sim_I h) \wedge (h' \vDash T))) \quad \Rightarrow \quad h \vDash T$$

That is, if T is true on every finite time interval, then it is true on the whole temporal axis.

To show the meaning of this definition better, let us use it to determine the safety of the two formulae: $SomF(P)$ and $WithinF(P, t)$, where P is a basic time-dependent formula.

First, we show that $WithinF(P, t)$ is a safety formula. To do so, let us consider its evaluation at generic time instant τ. Let us now consider a generic history h. If $h \vDash WithinF(P, t)(\tau)$ then there is nothing to show since the implication of the definition surely holds. If instead

$h \not\models WithinF(P,t)(\tau)$ then we must show that the antecedent to the implication is false. Since the formula is false for that history, it means that on the whole interval $I_\tau = (\tau, \tau + t)$ P is false. Now, take any finite time interval I such that $I \supseteq I_\tau$. Whatever h' that completes $h[I]$ we choose, it surely is $h' \not\models WithinF(P,t)(\tau)$, since the critical interval I_τ is already set to false. Hence the implication in the definition holds, so that we have shown the formula is a safety formula.

Now, we show that $SomF(P)$ is not a safety formula. Let us consider its evaluation at generic time instant τ. We just need to find a history h for which the antecedent of the implication is true and the consequent is false. Let us consider the history h such that P is false on the whole temporal axis. Obviously, $h \not\models SomF(P)(\tau)$. On the other hand, take any finite time interval I. Consider the following completion h' of $h[I]$: take any time instant t_τ such that $t_\tau > \tau$ and $t_\tau \notin I$ and make P to be true there. Obviously $h' \models SomF(P)(\tau)$ so that the implication is false. This shows that the formula is not a safety formula.

Both definitions 1 and 2 characterize safety in terms of *models*; in other words the characterization is *semantical*. On the other hand, the TRIO encoding in PVS, which we would like to use to carry out compositional reasoning, relies on purely *syntactical* inference rules.

In the previous sections we tried to formulate an equivalent definition of safety expressible as a TRIO formula. Unfortunately, it seems it is not feasible in a reasonably simple manner, nor formulating a condition subsuming safety and expressible as a TRIO formula seems feasible.

This means that we could not “transfer” the characterization of safety formulae, which is inherently semantical, to a syntax-based formalism.

As a result, there does not seem to be a simple and interesting use of safety formulae in expressing a rely/guarantee inference rule in TRIO. Nonetheless, the results obtained above about safety are still valid, and we will be able to formulate a sound and practical rely/guarantee inference rule, without the need to use them (see section 5.5).

5.4 A stronger semantics for rely/guarantee properties

In order to obtain an inference rule for composite systems which is both sound and simply expressible, we give a stronger semantics to a rely/guarantee specification. This means that instead of simple logical implication between temporally closed formulae we want to make stronger assumptions about how our specified system behaves with respect to an assumed behavior of its environment.

In the TLA temporal logic formalism (36), two operators are often used to formally describe systems in a rely/guarantee framework. These operators are usually represented as: $-\!\triangleright$ and $\stackrel{+}{-\!\triangleright}$, the second being a stronger version of the first one (i.e. if $\stackrel{+}{-\!\triangleright}$ holds then so $-\!\triangleright$ does, while the converse is in general not true). Their meaning in TLA is briefly explained on page 21.

Now, we propose something similar to those operators, but usable in the TRIO language.

Let P and Q be two time-dependent formulae, either primitive or built from primitive formulae using any TRIO operator. We define the $\overset{+}{\rhd}$ TRIO operator as a shorthand for the formula:

$$P \overset{+}{\rhd} Q \quad \triangleq \quad \begin{cases} AlwP_e(P) \Rightarrow AlwP_i(Q) \wedge NowOn(Q) & \text{if } \textit{Time} \text{ is dense} \\ AlwP_e(P) \Rightarrow AlwP_i(Q) & \text{if } \textit{Time} \text{ is discrete} \end{cases}$$

The operator $\overset{+}{\rhd}$ is usually called "while plus" operator and its symbol may be called "rely/guarantee arrow plus".

Informally speaking, the formula $P \overset{+}{\rhd} Q$ means that Q lasts at least as long as P does and even a bit longer, so that when P becomes false Q is still true for some instants (or more precisely for at least one time step, when the time model is discrete).

By considering this informal explanation of the meaning of the operators, we can understand that a rely/guarantee specification can be conveniently written as $E \overset{+}{\rhd} M$, where as usual E is the assumed behavior of the environment and M is the guaranteed behavior of the module being considered. Hence, the formula $E \overset{+}{\rhd} M$ means that:

1. As long as the environment behaves as in E the module guarantees a property M
2. If the environment stops behaving as in E the module can fail meeting its specification M only a bit later than the failure of the environment occurs

Moreover, note that the formula links the behavior of the environment and that of the module *since the beginning*, in that we consider a behavior where everything is correct since its start, that is at "time" $-\infty$ (or 0 if the time domain is natural).

It is obvious that any rely/guarantee specification must meet condition 1. Condition 2 is reasonable as well. In fact, every real system must be a causal one, that is must base the future behavior of its outputs on the present behavior of its inputs. Therefore, to fail condition 2 while respecting condition 1, a system must be able to stop respecting the property M at the same instant of time as when the environment stops respecting E. But, since the system bases its present behavior on the past behavior of the inputs and on its state, the only way it can do so is by predicting the future, which is clearly a non causal behavior. Hence, the use of the operator $\stackrel{+}{\rhd}$ to specify a rely/guarantee system should be a reasonable way to write specifications.

We now give some properties of the new operator $\stackrel{+}{\rhd}$.

Lemma 1 *For any formulae P and Q*

$$Alw(P \stackrel{+}{\rhd} Q) \wedge Alw(P) \Rightarrow Alw(Q)$$

Lemma 2 *For any formulae P, Q and R*

$$Alw(P \stackrel{+}{\rhd} Q) \wedge Alw(Q \Rightarrow R) \Rightarrow Alw(P \stackrel{+}{\rhd} R)$$

Lemma 3 *For any formulae P_i and Q_i, for $i = 1, \ldots, n$*

$$Alw\left(\bigwedge_{i=1}^{n}\left(P_i \stackrel{+}{\rhd} Q_i\right)\right) \Rightarrow Alw\left(\left(\bigwedge_{i=1}^{n} P_i\right) \stackrel{+}{\rhd} \left(\bigwedge_{i=1}^{n} Q_i\right)\right)$$

The proofs of lemmas 1, 2 and 3 are trivial and are omitted.

Now we enunciate and prove the following lemma.

Lemma 4 *For any formulae P, Q and R, if:*

1. $Som(AlwP_e(P))$

2. $Alw(Q \wedge R \Rightarrow P)$

then:

$$Alw(P \overset{+}{\rhd} Q) \;\Rightarrow\; Alw(R \overset{+}{\rhd} Q)$$

Proof for dense time models Assume the following to be true:

1. $Som(AlwP_e(P))$

2. $Alw(Q \wedge R \Rightarrow P)$

3. $Alw(AlwP_e(P) \Rightarrow AlwP_i(Q) \wedge NowOn(Q))$

4. $AlwP_e(R)$ at generic time instant t

Assumptions 1 and 2 are the two hypotheses of the lemma. Assumption 3 is the definition of the $\overset{+}{\rhd}$ operator for formulae P and Q and dense time models. Assumption 4 is instead a translation of the antecedent in the definition of the $\overset{+}{\rhd}$ operator for formulae R and Q. We want to prove that $AlwP_i(Q) \wedge NowOn(Q)$ holds at time t.

Assumption 1 can be equivalently rewritten as $AlwP_e(P)(u_0)$, that is by making explicit the time instant u_0 at which it holds. Let us now distinguish two cases, whether $u_0 \geq t$ or $u_0 < t$.

The first case: $u_0 \geq t$ is very simple. In fact, if we consider assumption 3 evaluated at time instant u_0, we can immediately conclude that $AlwP_i(Q) \wedge NowOn(Q)$ holds at time u_0. This also implies that $AlwP_i(Q) \wedge NowOn(Q)$ holds trivially for any time instant less or equal than u_0. Since $t \leq u_0$ by hypothesis, this case is concluded.

Let us now consider the case $u_0 < t$. The proof of this branch relies on what is a sort of induction done on temporal intervals instead of a discrete temporal domain. Very roughly speaking, starting from a condition true on a base interval, we propagate the condition using the assumptions, till we can show that this behavior is prolonged over the entire interval of interest. The basics of this technique have been proposed in (24) under the name "temporal induction" and have been implemented and proved in PVS in (20).

By assumption 1, we have a base point (u_0) on the temporal axis before which P is always true. So P is true on the open interval $(-\infty, u_0)$. By assumption 3, Q is also true on this interval, as well as in u_0 and for some more time $\epsilon_0 > 0$. So Q is true on the open interval $(-\infty, u_1)$ where $u_1 = u_0 + \epsilon_0$. Now, it is either $u_1 > t$ or $u_1 \leq t$. In the first case, the proof is concluded, since we have shown that Q lasts a bit longer than R does. If, instead, $u_1 \leq t$ we can use assumption 2 to conclude that P is true on the open interval $(-\infty, u_1)$ since on the same interval both Q and R are true. But this means that $AlwP_e(P)$ holds at time $u_1 > u_0$. This allows us to iterate the reasoning to get a new quantity $\epsilon_1 > 0$ so that Q is true on $(-\infty, u_1 + \epsilon_1) \supset (-\infty, u_0 + \epsilon_0)$. Now, by considering the sequence of points $u_1, u_2, \ldots, u_n$ where each u_i is defined as $u_{i-1} + \epsilon_{i-1}$, we can realize that we must eventually reach a point *after* t. In other words, there must exist a j such that

$u_j > t$. This in turn means that Q holds on the interval $(-\infty, u_j)$ and, consequently, that $AlwP_i(Q) \wedge NowOn(Q)$ holds at time t, thus concluding the proof.

It is important to note that there *must* exist such j and the series of points cannot accumulate before t. In other words, the value $u_0 + \sum_i \epsilon_i$ must eventually become bigger than t. This can be shown by contradiction: assume the opposite, that is there exists an infinite series of values ϵ_i such that:

$$u_0 + \sum_{i=0,\ldots,\infty} \epsilon_i = u_\delta < t$$

This implies that we can assure that Q holds till time u_δ only, that is $AlwP_e(Q)$ at time u_δ. Now, since $u_\delta < t$ by hypothesis, we can conclude that R holds as well till time u_δ. But then we can apply hypothesis 2 to conclude that P holds till time u_δ, or that $AlwP_e(P)$ at time u_δ. So, we use assumption 3 and conclude that there exists a positive quantity ϵ_δ such that $Lasts_{ie}(Q, \epsilon_\delta)$ holds at time u_δ. This contradicts the hypothesis that u_δ is the last time instant till which Q holds, thus showing that the series cannot converge to such a value.

Another way to explain temporal induction is by observing that this reasoning reduces to an ordinary induction over a discrete domain, where we induct on the sequence of points u_i. More precisely, the induction relies on the following scheme:

Base step At time u_0: $AlwP_e(Q)$

Inductive step At time u_i: $AlwP_e(Q) \;\Rightarrow\;$ At time $u_{i+1} > u_i$: $AlwP_e(Q)$

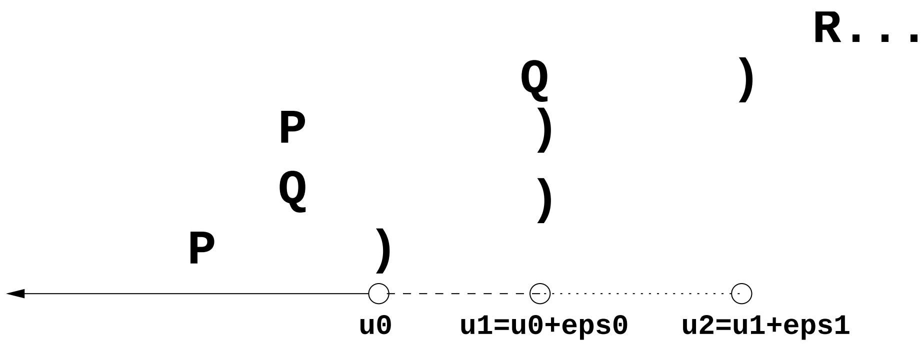

Figure 4. Induction on temporal intervals

Figure 4 gives some graphical intuition to the above reasoning. □

Proof for discrete time models Assume the following to be true:

1. $Som(AlwP_e(P))$

2. $Alw(Q \wedge R \Rightarrow P)$

3. $Alw(AlwP_e(P) \Rightarrow AlwP_i(Q))$

4. $AlwP_e(R)$ at generic time instant t

Assumptions 1 and 2 are the two hypotheses of the lemma. Assumption 3 is the definition of the $\overset{+}{\rhd}$ operator for formulae P and Q and discrete time models. Assumption 4 is instead a translation of the antecedent in the definition of the $\overset{+}{\rhd}$ operator for formulae R and Q. We want to prove that $AlwP_i(Q)$ holds at time t.

Assumption 1 can be equivalently rewritten as $AlwP_e(P)(u_0)$, that is by making explicit the time instant u_0 at which it holds. Let us now distinguish two cases, whether $u_0 \geq t$ or $u_0 < t$.

The first case: $u_0 \geq t$ is very simple. In fact, if we consider assumption 3 evaluated at time instant u_0, we can immediately conclude that $AlwP_i(Q)$ holds at time u_0. This also implies that $AlwP_i(Q)$ holds trivially for any time instant less or equal than u_0. Since $t \leq u_0$ by hypothesis, this case is concluded.

Let us now consider the case $u_0 < t$. If we define $u_i = u_0 + i$ for all $i = 0, \ldots, (t - u_0)$, it can be proved by induction on i. We apply the following induction scheme:

Base step At time u_0: $AlwP_i(Q)$

Inductive step At time $u_i < t$: $AlwP_i(Q) \;\Rightarrow\;$ At time u_{i+1}: $AlwP_i(Q)$

The base step follows immediately by combining hypotheses 1 and 3. Now, for the inductive step, assume $AlwP_i(Q)$ at time $u_i < t$. By hypothesis 4, and by the fact that $u_i < t$, it is also $AlwP_i(R)$ at time u_i. Hence, we apply assumption 2 to deduce that $AlwP_i(P)$ at time u_i. From the definition of $AlwP_i$, this is the same as saying that $AlwP_e(P)$ holds at time $u_i + 1 = u_{i+1}$. Now, we can consider hypothesis 3 at time u_i to deduce that $AlwP_i(Q)$ holds at time u_{i+1}.

The induction scheme allows us to conclude that $\forall i = 1, \ldots, (t - u_0)$ $AlwP_i(Q)$ holds at time u_i. In particular, for $i = t - u_0$ we have that $AlwP_i(Q)$ holds at time $u_{t-u_0} = u_0 + (t - u_0) = t$ which is exactly what we had to prove. $\square$

Hypothesis 1 in lemma 4 requires that P is true on a base interval, unbounded on the left. Whenever such a condition $Som(AlwP_e(P))$ holds for a formula P we say that P is "initialized", since we can guarantee that it is initially satisfied in the model we are considering.

5.5 A valid inference rule for rely/guarantee systems

We now have all the ingredients to formulate and prove a sound inference rule for rely/guarantee compositional reasoning. As discussed above, this rule is very similar to the invalid proposition 1 but with additional assumptions on the kind of semantics given to a rely/guarantee specification (i.e. use of the $\stackrel{+}{\rhd}$ operator) and on the initial conditions for the formulae representing the assumptions (i.e. we want them to be initialized). We give the following inference rule.

Proposition 3 (Valid rely/guarantee inference rule) *If, for $i = 1, \ldots, n$ (finite) the following conditions hold:*

1. $Som(AlwP_e(E_i))$*, that is* E_i *is initialized*
2. $Alw\left(E \wedge \bigwedge_{j=1,\ldots,n} M_j \Rightarrow E_i\right)$
3. $Alw\left(\bigwedge_{j=1,\ldots,n} M_j \Rightarrow M\right)$

then

$$Alw\left(\bigwedge_{j=1,\ldots,n} (E_j \stackrel{+}{\rhd} M_j)\right) \Rightarrow Alw(E \stackrel{+}{\rhd} M)$$

Proof If $Alw\left(\bigwedge_{j=1,\ldots,n}(E_j \overset{+}{\rightarrowtriangle} M_j)\right)$, then by lemma 3:

$$Alw\left(\left(\bigwedge_{j=1,\ldots,n} E_j\right) \overset{+}{\rightarrowtriangle} \left(\bigwedge_{j=1,\ldots,n} M_j\right)\right)$$

also holds.

Now, we note that $Som(AlwP_e(\bigwedge_{i=1,\ldots,n} E_i))$ holds. In fact, thanks to hypothesis 1, for each $i = 1, \ldots, n$ there exists an unbounded interval of the form $(-\infty, p_i)$ such that E_i is true on that interval. Hence, if we just consider the intersection of all those intervals $\bigcap_{i=1,\ldots,n}(-\infty, p_i) = (-\infty, \min_i p_i)$, the conjunction $\bigwedge_{i=1,\ldots,n} E_i$ holds on this derived interval.

Moreover, we note that $Alw\left(E \wedge \bigwedge_{j=1,\ldots,n} M_j \Rightarrow \bigwedge_{j=1,\ldots,n} E_j\right)$, since assumption 2 holds for every $i = 1, \ldots, n$.

Therefore, we can apply lemma 4 by substituting $\bigwedge_{j=1,\ldots,n} E_j$ for P, $\bigwedge_{j=1,\ldots,n} M_j$ for Q and E for R. We get:

$$Alw\left(\left(\bigwedge_{j=1,\ldots,n} E_j\right) \overset{+}{\rightarrowtriangle} \left(\bigwedge_{j=1,\ldots,n} M_j\right)\right) \Rightarrow Alw\left(E \overset{+}{\rightarrowtriangle} \left(\bigwedge_{j=1,\ldots,n} M_j\right)\right)$$

Finally, by assumption 3 and lemma 2 we get the desired result. $\square$

We want to point out that the formulae E, E_i, M and M_i can be temporally closed or not. If they are temporally closed, it simply means that the explicit universal closures with the Alw

operators found in the inference rule are just superfluous. However, it is likely that the more interesting cases are with non temporally closed rely/guarantee formulae.

We also give an immediate corollary to proposition 3 that may be straightforward to use with systems without a global environment assumption (such as closed systems).

Corollary 1 (Closed system rely/guarantee inference rule) *If, for* $i = 1, \ldots, n$ *(finite) the following conditions hold:*

1. $Som(AlwP_e(E_i))$*, that is* E_i *is initialized*
2. $Alw\left(\bigwedge_{j=1,\ldots,n} M_j \Rightarrow E_i\right)$
3. $Alw\left(\bigwedge_{j=1,\ldots,n} M_j \Rightarrow M\right)$

then

$$Alw\left(\bigwedge_{j=1,\ldots,n} (E_j \stackrel{+}{\rhd} M_j)\right) \Rightarrow Alw(M)$$

Proof From proposition 3, if we take $E = \texttt{true}$, we can conclude:

$$Alw\left(\bigwedge_{j=1,\ldots,n} (E_j \stackrel{+}{\rhd} M_j)\right) \Rightarrow Alw(\text{true} \stackrel{+}{\rhd} M)$$

Then, by applying lemma 1 substituting `true` for P and M for Q, the desired result follows. $\square$

What can we do whenever we are specifying a system where the inference scheme just proposed does not apply? Do we have to give up rely/guarantee reasoning completely and use only general purpose proof rules? A thorough answer would involve a long discussion and the analysis of several cases, which is out of the scope of this work. In this paragraph, we just sketch out the very basic developments.

Even though we did not investigate this issue in detail, it is possible that the rely/guarantee proof rule discussed above is not complete. This means that there are composite systems where some valid global properties cannot be proven with this rule, no matter how we apply it. It is important to have this in mind, so that we also know what are the possible ways out of it. The first one is to formulate different rely/guarantee proof rules to achieve completeness. This is possible and is shown in (46) with reference to temporal logic formalisms and the proof rules of (4). The major drawback is that complete proof rules cannot be applied mechanically like the ones seen in this chapter and require the prover to formulate some "auxiliary assertions" which usually cannot be done automatically, hence complicating the proof. The other way to achieve completeness is simply to use the basic proof rules of PVS (or any other general-purpose inference rules), which are complete, to achieve the goal, leaving out rely/guarantee proof rules.

On the other hand, we still believe that the rely/guarantee paradigm for writing modular specifications is useful in practice even when it cannot be fully applied. In fact, the paradigm still gives an interesting and often fruitful guideline in organizing a large specification, thus aiding the user in dividing the system in an effective manner. This organization often helps the verification process even if rely/guarantee inference rules cannot be applied, so it may still be useful in practice.

5.6 The complexity of compositional proofs

In this section, we want to analyze what are the benefits of adopting a rely/guarantee style for writing specifications, which allows the use of an *ad hoc* proof rule but requires to write a specification with stronger constraints. In order to do that, we analyze the proof of the theorem `rely_guarantee` in the composite class `two_echoers`, without using any rely/guarantee proof rule but relying entirely on the base proof commands of PVS. However, we are not going to describe the PVS proof steps in fine detail, but we just want to give a general idea of the overall structure of the proof. The reader who is uninterested can safely skip to the last paragraphs, where the conclusions are discussed.

The proof in PVS is rather simple in its essential steps, though not as short as one would expect. It relies on the following 2-steps induction scheme, which is directly derivable from the common 1-step induction scheme.

$$\forall P : \quad P(0) \wedge P(1) \wedge (\forall j \in \mathbb{N} : P(j) \Rightarrow P(j+2)) \;\Rightarrow\; \forall i \in \mathbb{N} : P(i)$$

where P is any predicate over naturals.

The basic sequent to be proved can be written as:

$$\begin{aligned} & (Alw(P_1.input) \Rightarrow Alw(P_1.output)) \\ \wedge \quad & (Alw(P_2.input) \Rightarrow Alw(P_2.output)) \end{aligned}$$

$$\vdash \overline{\qquad\qquad\qquad\qquad}$$

$$Alw(P_1.output) \wedge Alw(P_2.output)$$

Its proof can be split into two similar substeps. They are:

1 Prove that $Alw(P_1.output)$

2 Prove that $Alw(P_2.output)$

Since they are very similar, we will only describe the proof of subgoal 1, being the one of subgoal 2 easily derivable, using items of module `P2` instead of those of module `P1` and vice-versa.

The formula we want to prove represents the basic property of the system we want to prove: it always outputs `true`. To prove that, we use the induction scheme discussed above, so that we split to proof into:

1.1 Prove that $P_1.output$ at time 0

1.2 Prove that $P_1.output$ at time 1

1.3 Prove that $\forall j$: if $P_1.output$ at time j then $P_1.output$ at time $j + 2$

Subgoals 1.1 and 1.2 are the base step (which is two-fold in this particular induction scheme), while subgoal 1.3 is the inductive step.

Subgoal 1.1 is directly subsumed by the axiom `init` of class `P1`.

Subgoal 1.2 can be proved by using axioms `init` of class `P2` and `in_to_out` of class `P1`. In fact, by the first axiom we have $P_2.output$ at time 0. By the connections, this also means that

P_1.*input* at time 0. Hence, by axiom `in_to_out`, it is P_1.*output* one instant later, that is at time 1.

Subgoal 1.3 requires to prove that, if class `P1` outputs a `true` at generic time j, it will also output a `true` two time instants later. This can be done by using the axioms `in_to_out` of the two classes, which describe the basic output mechanism of the echoer. So, let us assume P_1.*output* at generic time j. By the connection axioms, this also means P_2.*input* at time j. Now, by axiom `in_to_out` for class `P2`, we deduce that P_2.*output* at time $j+1$. Using the connections again, this means that P_1.*input* holds at time $j+1$. Now we apply axiom `in_to_out` for class `P1` and deduce that P_1.*output* holds at time $(j+1)+1 = j+2$, so that subgoal 1.3 is concluded, thus concluding the whole proof.

How hard was this proof, done without use of rely/guarantee inference rules? The total proof was carried out by issuing 88 prover commands and consisted of 12 leaf sequents. This is definitely not a huge proof *per se*, however we should evaluate its complexity by relating it to the complexity of the specified system and the complexity of the property we want to prove.

The system is the composition of two equal modules (`P1` and `P2`) which are very simple, being described by only two short axioms. The property to be proven is also extremely simple, and its validity can be understood by a small amount of ordinary reasoning. Therefore, we believe that the proof is simply too long and complex to be acceptable. The same method with a slightly larger system, not to say a realistic-size system, is likely to carry poor results, in that the proof would be unacceptably large. Note that the complexity of the proof does not lie in the fact that it involves sofisticated techniques, since it is always clear what one has to do. The complexity lies entirely in the length and repetivity of the proof, since the same sequences of commands must be issued several times with some variations (e.g. different instantiations or different classes being considered).

The same proof, done with the system specified under the rely/guarantee paradigm, would instead involve just the application of the base proof rule, as discussed above. Hence, this would be a much simpler verification process, with fewer details to be considered.

All in all, there is (among many) a basic trade-off in specifying composite systems and proving their properties.

On the one hand we have freedom in writing specifications, so that we can express properties and formulae the way we believe it appropriate, concentrating mainly on ease of specification and on the clarity.

On the other hand we may want to sacrifice total freedom in writing a specification with respect to the ease with which we can carry out proofs. So, we may want to restrict the expressiveness of our models, choosing to adhere to a certain style in writing the specification, like the rely/guarantee paradigm. This may cause some more troubles in writing the specifications, but can provide much simpler proofs relying on tailored and powerful proof rules.

CHAPTER 6

AUTOMATED COMPOSITIONAL PROOFS

In chapter 4 we discussed how to translate a specification written in TRIO into the higher-order logic language of PVS (49). After that, we want to use the PVS proof checker to carry out computer-aided proofs of the system specified in TRIO, thus validating the specification or finding out errors, inconsistencies and flaws so that they can be promptly corrected.

The TVS (TRIO Verification System, sometimes also called TRIO/PVS) provides a mapping of the basic TRIO non-modular constructs onto PVS, together with a number of PVS proof strategies that automate several passages one often has to do when building a proof in PVS of an encoded TRIO specification. It was described in chapter 3. In this chapter we describe new proof strategies to be used with PVS encodings of modular TRIO specifications.

More precisely, a *PVS proof strategy* is a script, written in the LISP-alike PVS prover language (50), that describes a number of proof commands to be issued according to its parameters and current proof sequent. Once a strategy has been defined, it can be used as any other predefined proof command during the conduction of a proof.

The purposes of the proof strategies defined in the TVS system are basically two. First, we want them to be a shortcut for sequences of commands frequently applied sequentially during a proof, so that we do not have to type similar sequences of commands over and over with minimal changes; by doing this, proofs become shorter and also more readable. Second, we want them to hide the logic of the PVS system as much as possible from the point of view of the user building the proof, so that she/he does not have to know many details of the encoding but can reason as if the proof checker was built specifically for the TRIO language, except some minor details.

In this work, we focus mainly on realizing the first purpose, because it is more directly connected with the work on the mapping done in chapter 4 and it is of immediate use in shortening modular proofs, thus rendering possible the use of the modular features of the language with a certain ease. The second purpose would instead involve the implementation of so called *pretty-printing* PVS strategies, that are applied automatically by the system every time the sequent changes. They basically rewrite formulae in a nicer and simpler format to be read. These pretty-printing mechanisms already exist for TRIO in-the-small and can also be employed proficiently in modular proofs, since they basically rewrite basic TRIO formulae in a more readable way.

The proof strategies we designed are of two kinds. The first one is composed by strategies that make the use of the modular features of TRIO simpler and more automated. It is described in section 6.1. The second group is instead composed by strategies to be used with modular TRIO specifications in the rely/guarantee style. The need for these strategies arises naturally from the use of rely/guarantee proof rules in TVS. Hence, in section 6.2 below we first describe how to translate a rely/guarantee specification into the PVS language and we then describe the related proof strategies. Section 6.3 shows the benefits of the use of both rely/guarantee proof rules and of the described proof strategies in building a proof of a modular system similar to that described in chapter 5.

Finally, appendix B lists the full code of the PVS strategies described in this chapter, together with their syntax and a pseudo-code description of what they do.

6.1 Using modular TRIO in PVS

As discussed in chapter 4, TRIO classes are mapped onto PVS theories. Hence, we want to design proof strategies to handle two sets of common tasks in a modular proof: instantiation of formulae with proper parameters according to the class they belong to, and use of the connections to show the prover when two items are to be considered the same. These two aspects are discussed in the following two subsections.

Before doing that, we introduce now a very simple proof strategy that does not exploit any modular aspect of TRIO but is nonetheless rather useful in practice. In fact, it often happens that we need to manipulate a new formula by first expanding all the outer operators (typically, Boolean connectives) and finally flattening it so that it is distributed over new sequent formulae. As a simple example, consider the formula

```
(A!1 AND B!1)(tt!1)
```

We want to rewrite it into the two formulae

```
A!1 (tt!1)
B!1 (tt!1)
```

To handle such trivial but very often happening situations, we designed the strategy `open-fl`. Table VI in appendix B shows its simple pseudo-code description.

6.1.1 Strategies for class instantiations

We have seen, in chapter 4, that every TRIO class is translated into a PVS parametric theory. The first parameter of the theory is a non-empty type we usually name *instances* and is used when we import the theory into other theories to represent TRIO's use of modules. Moreover, every item of the base class is also parametric with respect to a constant of the given type *instances*. This is done to allow the definition of arrays of items of any TRIO class.

All this implies that, whenever we want to use that class into another class as a module, we first define a new non-empty type and we then import the parametric theory instantiating it with the new type. This allows a non-ambiguous naming of imported theories and also the importing of more than one instance of the same class, keeping them as separated modules as they are in TRIO.

When writing axioms and theorems for a given imported class, we usually want them to reference to all the elements of the array of identical items together, so that we use a universally quantified variable of type *instances* as the parameter of the items in the axioms and theorems. As a result of this practice, almost every formula is universally quantified with respect to a number of parameters of instantiation types.

Hence, whenever we prove such formulae in PVS, the first thing we do to make them usable is replacing the universal quantifications over parameters with Skolem variables. Then, each time we introduce other axioms or theorems in the same proofs, we usually instantiate their parameters of type *instances* with the same Skolem variables we have introduced at the beginning of the proofs, so that the prover knows we are referencing the same items, parametrically. This means that we have a noticeable number of new instantiations done with the same value, which results in typing the same instantiation commands over and over in various parts of the proof.

The strategies we consider in this section aim at simplifying this kind of situation. What we devised is a set of commands to repeat instantiations of new axioms or theorems by reusing instantiation values (i.e. Skolem variables or constants) introduced earlier during the proof. To do that, we first introduce a global variable that stores a hash table mapping arbitrary keys (i.e. either numbers or strings) to instantiation values, that is to a list of values to be used in instantiations of formulae. This global variable is initialized whenever the PVS LISP environment is first started and is changed only by the commands `set-def-inst` and `clear-def-inst` described below. It is deallocated only when the LISP environment is killed on exiting PVS.

Whenever we have an instantiation we think to be reusable during the proof, we issue the command `set-def-inst` to store its value into the hash table with a given key. Note that the value for the argument *key* has a default value of 0. This means that we have a sort of default mapping that is meant to store the most frequently used instantiation. This will typically be a full instantiation with Skolem variables for items of all the involved classes. The command is schematically described in Table VII in appendix B.

If we decide to clear all the current mappings and re-initialize the hash table, we can do so with the command `clear-def-inst`, shown in Table VIII in appendix B. Note that this command basically behaves as a `skip`, so that it is not stored into proof records since it does not change the proof sequent but only has side effect. On the other hand, the other command `set-def-inst` has been written so that it is recorded into proof reports even if it does not directly change the proof sequent, otherwise proofs which use this command would not be re-runnable correctly.

The first proof strategy that reuses strored instantiation values is `def-inst`. This simply does an instantiation (thus calling the `instantiate` command) on all the given formulae in the current sequent with the values for the given key. It is described in Table IX in appendix B.

As discussed above, default instantiation is often useful when we introduce axioms, lemmas or theorems in the current proof. Under this respect, the proof strategy `lm-def-inst` combines a lemma introduction with a default instantiation. It is described in Table X in appendix B. We point out one more thing about this strategy: the labelling of the newly introduced lemma. This simple thing is often very useful since it helps the user a lot in distinguishing which instances of lemmas appear in a given sequent. In fact, once a lemma is introduced we have no traces of which class it comes from, unless we label it with its full name. This helps a lot when we introduce formulae and we need to remember from which module we imported them from.

The strategy `lm-def-use` tries to take charge of the whole ordinary sequence of commands we usually issue when we introduce a lemma (i.e. an axiom or theorem) into a PVS proof of a modular specification. Therefore, it first introduces the lemma and instantiates its class parameters with pre-stored values. Then, it opens the external time quantification and tries to instantiate it according to the current context. This strategy may safely replace the usual sequences of commands we issue whenever we introduce a new formula, in that if any of its tasks fails, it simply skips it, so that in the worst case we end up with a simple introduction of a lemma. The strategy is described in Table XI in appendix B.

An example will show clearly how these proof strategies work in practice. Note that this example is not meant to be meaningful with respect to the system it describes, but is only introduced to show how the strategies behave.

Let us consider a base class `foo` which has two items named `it1` and `it2`. Another class `bar` imports two instances of `foo`, one with instantiation type `f1_type` and the other with instantiation type `f2_type`, as `F1` and `F2` respectively.

Let us also assume that class `bar` has the following two axioms and one theorem declared.

```
a1: AXIOM
Alw( F1.it1(f1) AND F1.it2(f1) )

a2: AXIOM
Alw( F1.it2(f1) IMPLIES F2.it1(f2) )

t1: THEOREM
Alw( F1.it1(f1) AND F1.it2(f1) AND F2.it1(f2) )
```

We start proving `t1` so that PVS shows the sequent:

```
t1 :

  |-------
{1}   FORALL (f1: f1_type, f2: f2_type):
        Alw(F1.it1(f1) AND F1.it2(f1) AND F2.it1(f2))
```

Obviously, we first of all introduce Skolem variables to replace the `FORALL` quantification. This leads to the formula to be proven:

```
{1}   Alw(F1.it1(f1!1) AND F1.it2(f1!1) AND F2.it1(f2!1))
```

We now want to set some instantiation values. More precisely, we want the default value (i.e. the one for key 0) to be the one corresponding to both variables `f1!1` and `f2!1`. We also want to keep an entry just for `f1!1` to be used in some other formulae. Hence we give the commands:

```
(set-def-inst ("f1!1" "f2!1"))
(set-def-inst ("f1!1") "one_only")
```

to store the two choices.

Now let us suppose we want to introduce axiom `a1` without opening its Alw() operator. We issue the command `(lm-def-inst "a1" "one_only")` so that the new formula introduced is:

```
{-1,(a1)}
      Alw(F1.it1(f1!1) AND F1.it2(f1!1))
```

with proper label and instantiation with the specified value.

Now, if we want to introduce axiom `a2` and also instantiate its Alw() operator at time `tt!1 + 4`, where `tt!1` is some valid Skolem variable, we issue the command `lm-def-use` getting the new formula:

```
{-1,(a2)}
      F1.it2(f1!1)(tt!1 + 4) IMPLIES F2.it1(f2!1)(tt!1 + 4)
```

which is ready for further manipulations and uses in the remainder of the proof.

6.1.2 Strategies for use of connections

The PVS axioms describing TRIO connections deserve a dedicated proof strategy to use them with ease during proofs. As discussed in chapter 4 a TRIO connection is mapped onto a PVS identity. Let us consider an item A connected to an item B in the current class. Let us assume we are building a proof that wants to state some property of the item B. It often happens that we get to a proof of the same property but for item A. At that time, we want to use the definitions of the connections, to tell the prover that the proof is really concluded, thanks to the identity we are using. To do that, we need to introduce the axioms describing the connections and instantiate them with proper instantiation values, if needed. Then, if the situation is the one just described, with the same formula among the antecedents and among the consequents, except for a connection identity, a `grind` command will simply close the sequent, by automatically applying the rewritings (i.e. the substitutions) induced by the identities. Usually, the sequent is closed with less powerful commands as well, but they are simply subsumed by a `grind` which is more likely to succeed.

One may think that the use of a dedicated strategy to introduce connections is not necessary, and can be safely replaced by the use of auto-rewrites declarations in the PVS theory. Informally, a *rewriting rule* is an equality which can be used autonomously by the PVS prover. More precisely, if the equality has the form $LHS = RHS$, whenever PVS has a term of the form LHS it knows it can replace it with RHS and vice-versa. The PVS language has a particular directive (`AUTO_REWRITE`) that, whenever put in a theory declaration, tells the prover that, as soon as it has started, it has to load the rewriting corresponding to the specified formula. So, we may think that if each connection axiom had an auto-rewrite declaration associated with it, the prover would autonomously recognize when it can be used. Unfortunately, this is in general not true. The problem is that the connection axioms must be instantiated with the instantiation parameters before being used. The PVS heuristics usually fail in determining the right instantiation values, so that some user interaction is needed to choose the right actuals.

Therefore, we designed the `connect` proof strategy, described in Table XII in appendix B. It combines the basic sequence of commands described above with the knowledge of how connection axioms should be named, according to the rules defined in chapter 4. So, if the connection axiom is just one, it is called *connections*, while if there are N connection axioms they are named *connection_i* for $i = 1, \ldots, N$. The strategy introduces all the connection axioms it finds in a given class (parameter *prefix*) and provides default instantiations as needed. It then tries to close the sequent with a `grind` if the user wants that.

As a very simple example to see the use of this strategy, consider the classes `foo` and `bar` described in the previous section. Now assume `bar` has the following two connection axioms.

```
connection_1: AXIOM
connect(F1.it1(f1), F2.it1(f2))

connection_2: AXIOM
connect(F1.it2(f1), F2.it2(f2))
```

Hence, if we have a sequent like the following

```
[-1]  F1.it1(0)(tt!1)
[-2]  F1.it2(0)(tt!1)
  |-------
[1]   F2.it1(f2!1)(tt!1) AND F2.it2(f2!1)(tt!1)
```

it can be closed by means of the connections. In fact, if the default instantiation has been set to `(0 f2!1)`, the sequent is closed by the command `(connect t)`.

6.2 Rely/guarantee proofs in PVS

In this section we want to discuss some details about how a modular system specified in TRIO according to the rely/guarantee paradigm, introduced in chapter 5, can be proficiently translated in PVS, in order to exploit the rely/guarantee proof rule of proposition 3, together with the automated proof capabilities of the PVS environment.

6.2.1 Encoding of a rely/guarantee specification

The basic modular description of a TRIO class in PVS is just the same as discussed before. Now, we need to add the implementation of the rely/guarantee operator and a statement of the proof rule usable in practice. To do that, we define a new PVS theory named `TRIO_relyguarantee` parametric with respect to a natural number `N`. This parameter represents the number of modules we are composing, so it is basically the same n as that of the rely/guarantee proof rule of proposition 3. Therefore, let us suppose we are translating a TRIO class C into a PVS theory with the same name. If C is composed by k modules and we want to prove results about this composition, we first of all need to specify an importing clause `IMPORTING TRIO_relyguarantee[k]`. The same theory must be imported in the PVS description of each of the submodules, since it contains the definition of the operator $\overset{+}{\rhd}$. In this case, we may give an arbitrary value for the parameter `N`, for instance a 0.

The operator $\overset{+}{\rhd}$ is implemented in the theory `TRIO_relyguarantee` as the PVS infix operator `>>=`. Note that the symbol `>>=` has been chosen only because it is a PVS infix operators available to the user for redefining, and any other meaning it may have because of other declarations in other theories is just coincidental. Obviously, its definition is just a translation of the formal definition of the operator in TRIO.

```
t: VAR Time
E, M: VAR TD_Fmla

%\rgarrowplus operator
>>=(E, M)(t): boolean = ( AlwP_e(E)(t)
                   IMPLIES (AlwP_i(M)(t) AND NowOn(M)(t)) )
```

Notice that a more natural way to translate it would have been as a time-dependent formula instead of an explicit function of time, thus relying on the corresponding TRIO operators for time-dependent formulae. So, `>>=` could have been equivalently written as:

```
>>=(E, M): TD_Fmla = AlwP_e(E)
            IMPLIES (AlwP_i(E) AND NowOn(M))
```

However, the definition we used showed to be a lot better in practice when used during PVS proofs, since it requires a smaller number of opening and rewritings. In fact, it often makes proofs much shorter.

The next thing to denote in a rely/guarantee specification is when a given formula is initialized. To achieve this, we introduce a subtype of time-dependent formulae, named `Initialized_Fmla`. An `Initialized_Fmla` is simply a `TD_Fmla E` for which $Som(AlwP_e(E))$ is true.

```
%Initialized formula
Initialized?(E): boolean = Som( AlwP_e(E) )
Initialized_Fmla: TYPE+ = { E | Initialized?(E) }
```

Now, we can write the statement of the rely/guarantee proof rule seen in proposition 3 as a PVS theorem. Since we have to relate the behavior of `N` subclasses with the one of the upper level class importing them, we need an extension of the TRIO operator $\stackrel{+}{\rhd}$ to handle the N classes altogether. More precisely, we first of all define a range type to enumerate all N subclasses:

```
rng: TYPE+ = {i: nat | 0 < i AND i <= N} % 1..N classes
Rng_Fmla_Type: TYPE+ = [rng -> TD_Fmla]
```

Now, we define an extension of the $\stackrel{+}{\rhd}$ operator to predicate on variables of `Rng_Fmla_Type` instead of simple time-dependent formulae. So, we introduce the following definitions.

```
j: VAR rng
P_i, Q_i: VAR Rng_Fmla_Type

>>=(P_i, Q_i): Rng_Fmla_Type = (LAMBDA j: (P_i(j) >>= Q_i(j)))

rwrt: FORMULA
(P_i >>= Q_i)(j)(t) = (P_i(j) >>= Q_i(j))(t)

AUTO_REWRITE+ rwrt
```

The directive `AUTO_REWRITE+` tells PVS to autonomously activate this rewriting rule whenever the current theory is imported in another; this is of great help during proofs.

Then, we introduce four variables to represent the formulae E, M, E_i, M_i ($i = 1, \dots, n$) and we name them `E_g`, `M_g`, `E_i` and `M_i` respectively (the subscript `g` stands for "global"). In particular, we require the formulae `E_i` to be initialized.

```
E_i: VAR [rng -> Initialized_Fmla]
M_i: VAR Rng_Fmla_Type
E_g, M_g: VAR TD_Fmla
```

Finally, we can formulate the rely/guarantee inference rule of proposition 3 by using the operators we have just defined.

```
Rely_Guarantee_inference_rule: THEOREM
  ( Alw( E_g AND FA(M_i)  IMPLIES  FA(E_i) )
         AND        Alw( FA(M_i)  IMPLIES  M_g )
         AND        Alw( FA(E_i >>= M_i) ) )
IMPLIES
Alw( E_g >>= M_g )
```

We want to point out three things about this implementation to explain it better. First, the `FA` operator is a universal quantification over a variable of type `rng` and it comes from the importing of the basic TVS theory `trio_quantif[rng]`. Second, the requirement that `E_i` is initialized is implicit in the statement of the theorem, since `E_i` has been declared as a range of initialized formulae. This will result in a TCC (Type Correctness Constraint) generated during the instantiation of the term `E_i` requiring to prove that the actual replacing `E_i` is an initialized formula. Third, the first hypothesis of the theorem simply rewrites the hypothesis to proposition 3

$$\bigwedge_{i=1,\dots,n}\left(E \wedge \bigwedge_{j=1,\dots,n} M_j \Rightarrow E_i\right)$$

in the equivalent form

$$E \wedge \bigwedge_{j=1,\dots,n} M_j \Rightarrow \bigwedge_{i=1,\dots,n} E_i$$

Before describing the proof strategies for rely/guarantee proofs, we still need to say something about what is a convenient way to carry out rely/guarantee proofs in PVS using the above declarations and definitions. In fact, when one wants to employ the proof rule seen above, she/he has to introduce its statement in the proof sequent and then to instantiate its variables referring to what `E_g`, `M_g`, `E_i` and `M_i` are in the specification. We think a convenient way to do that is by means of four definitions and four axioms, so that they can be promtly used during proofs. If we have a class adopting compositional reasoning on `k` subclasses, we should introduce the following declarations.

```
E: TD_Fmla
E_def: AXIOM
E = ...

E_i: Rng_Fmla_Type[k]
E_i_def: AXIOM
(E_i(1) = ...) AND (E_i(2) = ...) AND ...
   AND (E_i(k) = ...)

M: TD_Fmla
M_def: AXIOM
M = ...

M_i: Rng_Fmla_Type[k]
M_i_def: AXIOM
(M_i(1) = ...) AND (M_i(2) = ...) AND ...
   AND (M_i(k) = ...)
```

The proof strategies we are now discussing rely on this conventional naming for the declarations.

Similarly to what happens with other naming conventions for base TRIO, as discussed in chapter 4, we realize that the traslation from a TRIO class to its corresponding PVS mapping can be fully automated by developing adequate translation tools. In fact, this is what is currently being developed in a tools suite for the TRIO language. Hence, the above instructions on how to translate a rely/guarantee specification from TRIO to PVS are not meant to be followed directly by a human user, since it would be extremely annoying and time-consuming, but must be considered with respect to an automatic translation support that, once more, lets the user concentrate on the TRIO language rather than the encoding details in PVS.

6.2.2 Strategies for rely/guarantee proofs

Just like in a generic modular proof we often need to use the axioms describing the connections, in a rely/guarantee proof we often need to use the definitions for the formulae E, M, E_i and M_i. We can implement this use into a PVS strategy very easily, by using the previously seen strategies for default instantiations of lemmas. In fact, assuming correct instantiation values have been stored, and that the axioms defining E, M, E_i and M_i have been named as discussed in the previous paragraph, we just need to call the strategy `lm-def-use` four times. This extremely simple strategy is named `rg-use-definitions` and is described in Table XIII in appendix B. Note that, for the same reasons as with the connection axioms described above, an auto-rewriting is not enough to handle these axioms effectively, so that a proof strategy is needed.

Now, we want an *ad hoc* strategy to deal with the parametric representation of the $2N$ formulae E_i and M_i. In fact, they are represented in PVS as a mapping from an index ranging in the interval $1, \ldots, N$ to time-dependent formulae. Since the axioms and theorems of the subclasses we use during a proof refer to a single formula, we usually have statements representing predicates of the form $P(E_i(i!1), M_i(i!1))$, that is where we represent the parametrization with respect to the index i with a Skolem variable (in fact, $i!1$ is meant to be of type `rng[N]`). What happens during the (usually) last steps of a subgoal for a given rely/guarantee proof is that we have to distinguish the cases for each $i = 1, \ldots, N$. In simpler situations, that is typically when the N submodules are all of the same class, we can conclude each case with the same sequence of commands. In more general situations each case may require its own dedicated proof. However, the strategy `rg-i-case` handles the splitting of a proof sequent into N subgoals for each value of the given *var* argument ranging from 1 to N included. The strategy also tries to close each generated sequent with the usual `grind` heuristic, unless required not to do so. This strategy is schematically described in Table XIV in appendix B.

6.3 A comparative example of modular proof

In this section, we want to illustrate an example proof of a composite system. We are going to do the same proof first without use of any of the proof strategies described above and without use of the rely/guarantee paradigm and proof rule. Then, we will redo the same proof using both the new strategies and the rely/guarantee proof rule. We want to show the differences of the two approaches, the different organization of the proofs and the benefits of adopting the rely/guarantee proof rule and in using the proof strategies in terms of readability, simplicity and length of the resulting proofs.

In order not to expose the reader to too many details, the proofs are described very synthetically in this chapter, just to convey a general idea and to be able to draw comparisons and conclusions. However, a longer and much more detailed report of the proofs is available in appendix C.

6.3.1 System description and specification

The system we work on is similar to that described in chapter 5. The main difference is that we now adopt a continuous time model. We know TRIO can be used with any time model, but we need to remind that the present TVS implementation for TRIO in-the-small handles the continuous time case only. In continuity with the current implementation, we focus on a continuous model as well. Another difference in this new system with respect to the one of chapter 5 is in the definition of the base axiom `in_to_out` of the class `echoer`. Let us consider the new class, named `echoer_rg`.

```
class echoer_rg

        signature:

                visible:
                        input, output;

                temporal domain: real;

                items:
                        state input;
                        state output;

        formulae:

                axiom init:
                AlwP_i(output)(0);

                axiom in_to_out:
                input -> output & Lasts_ii(output, 1);

end
```

As it is obvious from the axiom `in_to_out` the system is somewhat underspecificated. In fact, the axiom describes the behavior of the class when `input` is true but does not specify it when `input` is false. This is not a problem with respect to our goals; on the contrary, it renders the system a bit simpler both in its description and in the proofs, so that it is more profitable as an example. Another feature of the specification that may seem annoying is the redundancy in the same axiom. In fact, obviously $Lasts_{ii}(item, t) \Rightarrow item$ so that we could drop the first term of the conjunction. However, once again this redundant notation is going to render the proofs a bit simpler in some passages, so we will adopt it, even if it may be considered inelegant.

Let us now consider the class `two_echoers_rg` which is all identical to the previously seen class `two_echoers`, except for its time model. So, here is its definition in TRIO.

```
class two_echoers_rg

        import:
                echoer_rg;

        signature:

                temporal domain: real;

                modules:
                        P1, P2: echoer_rg;

                connections:
                        (direct P1.output, P2.input);
                        (direct P2.output, P1.input);

        formulae:

                theorem Rely_guarantee:
                P1.output & P2.output;

end
```

We now analize the two proofs of the same theorem, in the two following sections.

6.3.2 Proof without strategies and rely/guarantee proof rule

The proof of the `Rely_guarantee` theorem is too long to be done in a single shot, so that we need to enunciate and prove some auxiliary lemmas to encapsulate fundamental steps in the proof and introduce them when needed to get to the final result.

More precisely, the proof of the result Alw(P1.output) can be split into the following steps:

1. $AlwP_i$(P1.output) at time 0
2. $AlwF_e$(P1.output) at time 0, by proving that:
   (a) $Alw(\text{P1.input} \Rightarrow Futr(\text{P1.input}, 1))$
   (b) P1.input at time 0

Obviously, we have a similar situation for the item P2.output, except that we consider items of the class P2 instead of P1.

The above proof scheme leads naturally to the following intermediate results to be proved within the class `echoer_rg`.

```
theorem now_and_nexttime:
( Alw(input -> Futr(input, 1)) & input(t) )
        -> (all i: Dist(input, i))(t)

theorem alw_output:
( Alw(input -> Futr(input, 1)) & input(t) )
        -> AlwF_i(output)(t)
```

where `t` is a variable of type Time (i.e. real) and `i` is of type natural.

The first theorem basically says that if condition 2.a holds and there is an initial time instant in which input holds, then input holds at every integer time unit. The second theorem extends the first one, in that it says that, under the same conditions, output holds at every time in the future.

Let us now consider the proofs of the two theorems. We just sketch out the proofs in this section; all the details are available in appendix C.

The proof of theorem `now_and_nexttime` is really simple and relies on induction on variable `i`. Both the base case and the inductive step can be discharged by using the antecedents of the implication combined with the inductive hypothesis.

The proof of theorem `alw_output` is basically divisible in two parts. The first part is to prove `Dist(output, u)(t)` when `u` is a natural number. This reduces to the other theorem `now_and_nexttime`. The second part is to prove `Dist(output, u)(t)` when `u` is not a natural number. In this case we take the *floor* of `u`, apply theorem `now_and_nexttime` to it and then "propagate" the output till the desired time by means of axiom `in_to_out`, which guarantees a duration of 1 time unit.

Before being able to prove the main theorem, we still need to prove another intermediate result, that is we have to show that step 2.a above holds for class `two_echoers`.

```
theorem rg_aux:
P1.input -> Futr(P1.input, 1)
```

and the analogous for module `P2`. The formula is proven by using the axioms `in_to_out` from both modules `P1` and `P2`. This allows us to show that:

$$\text{P1.input} \Rightarrow \text{P1.output} \Rightarrow \text{P2.input} \Rightarrow Futr(\text{P2.output}, 1) \Rightarrow Futr(\text{P1.input}, 1)$$

Note that the second and fourth implications are derived by using the information available from the connections in the global class.

After these preparatory theorems, we can get to the global result, that is `Alw(P1.output)` and `Alw(P2.output)`. We distinguish the cases for time $t \leq 0$ and $t > 0$. The first case is simply subsumed by the initialization axioms `init` of the two classes. The other case requires the intermediate lemmas `alw_output` and `rg_aux` to be proved, together with the information about the connections.

6.3.3 Proof with strategies and rely/guarantee proof rule

Now, we are going to build a different proof of the same global property of the class `two_echoers_rg`, where we use both the PVS strategies described in sections 6.1 and 6.2, and the rely/guarantee paradigm to specify the system and build the proof. As usual, we only sketch out the proof here, while all the details are available in appendix C.

We do not have to devise a division of the proof into intermediate auxiliary lemmas, since the rely/guarantee paradigm already prescribes how to divide the proof among the classes. More precisely, we have a property to be proven which is local to the two modules `P1` and `P2` and is simply described by the rely/guarantee formula $input \overset{+}{\rhd} output$.

After proving that, we can build the proof of the global property `Rely_guarantee` using this theorem together with the general proof rule for rely/guarantee systems.

To prove $input \overset{+}{\rhd} output$ we need to expand the $\overset{+}{\rhd}$ operator into its definition and prove it. Hence, we reduce to the substeps:

1. $AlwP_e(input) \Rightarrow AlwP_i(output)$

2. $AlwP_e(input) \Rightarrow NowOn(output)$

Subgoal 1 relies on axiom `in_to_out`, exploiting the consequent $Lasts_{ii}(output, 1)$ to show that `output` also holds at the current time instant. Subgoal 2 is also similarly proven, that is we use the derivable term $Lasts_{ii}(output, 1)$ to show that there exists a non-empty time interval in the future over which `output` holds.

The proof of the global property $Alw(P1.output \wedge P2.output)$ is done by using the rely/guarantee proof rule of proposition 3. More precisely, we have to show that the following hypotheses hold:

1. P1.output $\wedge$ P2.output $\Rightarrow$ P1.input $\wedge$ P2.input
2. P1.output $\wedge$ P2.output $\Rightarrow$ P1.output $\wedge$ P2.output
3. P1.input $\overset{+}{\rightarrowtriangle}$ P1.output
4. P2.input $\overset{+}{\rightarrowtriangle}$ P2.output
5. P1.input and P2.input are initialized

Now, condition 2 is trivially true. Condition 3 and 4 are just the rely/guarantee theorems for the modules `P1` and `P2` which we have already proven. Condition 1 is deducible by using the information associated with the connections. Condition 5 is a consequence of the axiom `init` for modules `P1` and `P2` and of the connections of the system. Then, the rely/guarantee proof rule allows us to conclude the global property holds. Building the proof in PVS requires indeed to cover some more technical details, that are shown in appendix C.

6.3.4 Comparison of the two proofs

In this section we want to draw a comparison between the two proofs of the same property, that is the proof done without using the rely/guarantee proof rule and the new PVS proof strategies described in section 6.3.2, against the proof done using the rely/guarantee paradigm and the PVS strategies introduced in this chapter, listed in section 6.3.3.

We want to compare the two proofs in terms of length, of intricacy, of modularization of the steps and of how readable and simple to follow they are. Let us try to give a more precise characterization of the features of a proof we have just listed.

We introduce a metric for the length: we measure the length of a proof in terms of the number of PVS commands we issue in the prover environment to conclude the proof. Since it is common that the proof of a theorem requires the introduction of other previously declared theorems, lemmas and axioms, we should count the length of the proofs of those as well. Otherwise, we could make any proof to be of minimal length by redeclaring and proving our theorem under another name, and then introducing it to get to the conclusion immediately.

To introduce, in the main formula, a simple representation of the dependencies in the proof, we use a directed graph. Each node in the graph represents a lemma or theorem used in the proof of the formula we are considering, including the formula itself. We exclude from the nodes the axioms and the definitions, since they do not require a proof themselves. There is an arc from a node N_1 to a node N_2 if and only if N_1 requires N_2 in its proof, that is there is some command for lemma introduction (e.g. `lemma`, `forward-chain`, `use`, `lm-def-use`, etc.) with N_2 as argument. Each arc is weighted with the number of times N_1 has introduced N_2 in its proof. Moreover, we store in each node a number representing the number of proof commands the proof of that node has required directly in PVS.

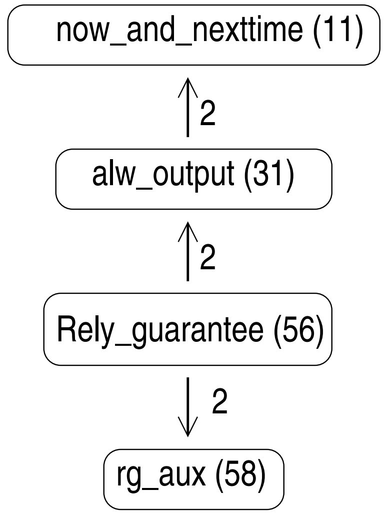

Figure 5. Proof dependencies in normal proof

In Figure 5 we have the dependencies graph for the proof done without rely/guarantee proof rule and without new PVS strategies as described in section 6.3.2.

In Figure 6 we have instead the dependencies graph for the other proof with rely/guarantee proof rules and new PVS strategies, described in section 6.3.3. Note that the rely/guarantee proof rule which is used has not been considered as an auxiliary lemma, since it is a powerful rule proved once and for all and may be considered like an additional extended propositional rule like those PVS has built-in.

Once the graph representing the dependencies for a given proof has been built, we can give two measures of how long the proof is. The first, simpler measure just sums up the values

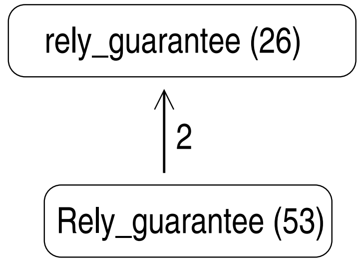

Figure 6. Proof dependencies in rely/guarantee proof

contained in each node, thus counting the total number of proof commands typed in by the user to get to the final result. We call this value "modularized proof length".

This measure, however, does not take into account the effort made to break the proof into several lemmas and how this division is done: this basically means how well the proof was modularized and more precisely if the intermediate lemmas are sufficiently independent and well chosen. Measuring all these things is obviously rather difficult and is also somewhat a subjective matter. However, we introduce a different metric on the graphs that tries to take some of these issues into account. More precisely, we want to try to give a comparative metric which tells how many proof commands we should approximately issue if the proof was done in one single shot, without identifying and using auxiliary lemmas of any kind. So, every time we introduce a lemma we should count as if we immediately prove it before its use. Hence, if the same lemma is introduced more than once, we should count its weight every time, as if we had to prove it again and again. By doing this, if the division into lemmas was well done, and more precisely if the parts of the proofs are sufficiently well decoupled and independent, then the auxiliary lemmas are not introduced too many times during the main proof, thus keeping this count acceptably low. In order to be more formal, we notice that the dependencies graph is by definition an acyclic graph. Hence, we can define an ordering of the nodes which is compatible with the topological order relation induced by the arcs. So, starting from the leaves we calculate the value for each node, given by the sum of the current value in each child node times the weight of the arc which goes to that child, for every node among the children of the curren one. We do not explain this calculation in more detail, since we are confident that every reader who has some familiarity with simple graph optimization algorithms will easily understand it. We call this metric “monolithic proof length” since it is an estimation of the length of a single proof encompassing all the intermediate results.

Table IV compares the results in terms of length in the two proofs we are considering, with both metrics.

PROOF	Modularized proof length	Monolithic proof length
Normal proof	156	278
Rely/guarantee proof	79	105
Improvement	197 %	265 %

TABLE IV

Comparison of length of the two proofs (in proof commands)

It is clear from this data that the use of the rely/guarantee proof rule, together with the new PVS proof strategies has greatly shortened the proof, reducing it to one of a simply manageable size. Since the class of which we have proven the global property is rather simple to describe informally, we expected a similarly acceptably short and simple proof.

We can introduce another simple metric to evaluate the complexity of a proof: the number of leaves of the proof tree. Every PVS proof can be represented by a tree: each node in the tree is a proof command. Starting from the base sequent, whenever the command causes the current goal to be split into n subgoals, the node in the tree has n children. Recursively, the proof of each of the subgoals is a subtree rooted at the corresponding branch. PVS can generate automatically these trees from a proof. A measure on these trees which may be considered as an indication of the complexity of the proof is the number of leaves. In fact, counting the leaves is the same as counting how many distinct subgoals had to be proven. Usually, the fewer the subgoals a proof is split into, the simpler the proof is to follow. Of course this may be not always true. For example, if the splitting into subgoals corresponds to strongly independent parts of the proof, it may even divide proficiently a long proof, thus rendering it simple to be understood. However, most of the times if the leaves are too many it probably means that the proof was rather hard to follow. Table V compares the number of leaves for the two proofs of the global property.

Lemma	Number of leaves
echoer_rg.now_and_nexttime	2
echoer_rg.alw_output	4
two_echoers_rg.rg_aux	4
two_echoers_rg.Rely_guarantee	8
Total for normal proof	18
echoer_rg.rely_guarantee	3
two_echoers_rg.Rely_guarantee	5
Total for rely/guarantee proof	8

TABLE V

Comparison of leaves of the two proofs

Also according to this metric, the proof using the rely/guarantee proof rule and the PVS strategies is visibly simpler than the other one, also because it involves a smaller number of intermediate lemmas (and namely just one).

Besides these numerical comparison of the two proofs, we can also give some informal comments about how different they were to be completed. The first thing to notice is the different use of modularization in the proofs, that is the way they have been divided into intermediate lemmas.

The basic fact is that the rely/guarantee paradigm has given a basic layout to organize the proof into lemmas. More precisely, we know that each class must have a local rely/guarantee property. This property can be proven locally, that is by using axioms and items of the class only. By composing these local lemmas, we build up the global property we want prove, and use a predefined proof rule to carry out the final result. This proof rule helps a lot in dividing the global proof into subparts, one for each of the hypotheses in the rely/guarantee proof rule. This proof rule also defines clearly how the local properties should be used into the global proof and how are to be combined. All these guidelines greatly help in building the desired proof, since we have less things to consider and we can rely on a rather powerful method. Moreover, the rely/guarantee paradigm helps a lot in building a proof that is well modularized, in that each of the intermediate lemmas we build and prove is well decoupled from the others and from the global statement. This can be seen, among other things, by the fact that the dependencies graph of Figure 6 is simple, so that we do not have to use the same intermediate result over and over during the proof. On the contrary, the intermediate lemma only represents a sort of "macro step" of the proof encapsulated nicely: it basically encompasses the proof of what can be proved locally.

On the other hand, the normal proof required the user to "invent" from scratch intermediate lemmas, according to her/his intuition and experience on how to carry out the proof. This surely is an advantage in terms of flexibility and freedom of specification, a quality often sought for. However, it is true that this also implies that building the whole proof results longer, more difficult and time-consuming. Whenever a large flexibility and freedom of specification is not really needed, it is usually better to exchange them with a smaller effort to complete a correct specification and verification. Also notice that the modularization of the proof represented in Figure 5 shows an higher coupling between the lemmas of the proof, since the weights on the arcs are always greater than 1. This may indicate that another, better modularization is probably possible, even if the one we used is intuitive and rather neat.

All in all, we think that it is absolutely clear that using the rely/guarantee paradigm has brought many benefits with respect to the ease in carrying out the proof. Together with that, the new PVS strategies have also played an important role, encapsulating recurrent routinary sequences of commands and closing in a few commands sequents that would have otherwise required many more. The rely/guarantee paradigm is also useful *per se* in guiding the specification of the system, at least in those cases when it is applicable without loss of generality or whenever a total freedom in writing the specification is not needed.

CHAPTER 7

CONCLUSIONS

This work constitutes an attempt to concretely design a compositional framework to specify and verify large modular systems under the rely/guarantee paradigm and with reference to the TRIO specification language. More precisely, we basically achieved the following results.

We provided an effective mapping of the modular features of the TRIO language onto PVS, based on the current encoding for the non-modular features of TRIO. Based on this mapping, a number of proof strategies to automate frequently occurring passages of compositional proofs were designed and made available as a part of the support tool TVS.

We introduced the rely/guarantee paradigm in TRIO, giving general guidelines on how a composite rely/guarantee specification should be written and also proving a proof rule to infer properties of composed modules. We believe that this proof rule is not only of methodological interest but also of practical usefulness.

To show this, we made the TRIO rely/guarantee proof rule supported in PVS, as another extension of the basic TVS. The use of these extensions showed to bring strong benefits in conducting some example compositional proofs. We believe the benefits will surely increase as the systems under analysis become larger and more complex.

As a final remark, we point out that no technique, method or language for the specification and verification of systems is likely to be a magic bullet. The formal analysis of large systems is an unavoidably hard task, because of the inherent complexity of the systems; therefore no technique is likely to make it become trivial. However, we firmly believe that the application of modularization techniques, the use of neat specification languages, the adequate support of analysis tools and a constant practice can surely make the analysis of large systems a practically doable task, thus ensuring the development of reliable industrial-size real-time systems.

7.1 Future work

While achieving its basic goals, this thesis also suggested several new directions for future work. Let us briefly consider the most interesting of these suggestions.

The need for automatic translators from TRIO to PVS is clear. In fact, the details of the mapping of all the features of the TRIO language are usually uninteresting for the user who should deal directly with TRIO code only. In particular, the translators (better if in the form of integrated editing environment to develop large specifications) would be responsible for the correct management of visibility and inheritance into PVS, which is at the moment largely deficitary.

Another aspect of the same problem of hiding the details of PVS as much as possible from the TRIO user arises during the conduction of automated proofs. Pretty-printing strategies should then be developed to show a proof environment as much “TRIO-like” as possible, including a coherent representation of the importing hierarchy between modules.

Altough the TVS tool is a fundamentally prototypal tool, a TRIO open and integrated environment is currently on the way. It will encompass front-ends to model checkers and theorem provers, test-case generators, etc. When in its maturity, this project will surely fulfill many of the needs discussed above.

Another interesting direction for future work is an accurate analysis of the completeness of the compositional rely/guarantee proof rule given in chapter 5. We speculated it is probably not complete, but this important feature deserves more investigation to draw provable results. Moreover, in case the proof rule showed to be incomplete, efforts should be made to see how and if it can be made complete, or if other, more general rely/guarantee proof rules can be formulated.

Another interesting analysis could be what we may call "semantical characterization of formulae". In fact, in chapter 5 we discussed safety of formulae and showed how it is impossible to give a completely syntactical characterization of safety in TRIO. This issue could be analyzed in more detail, for example to see if alternative (non equivalent) definitions of safety are possible in TRIO and if they can be of any use in defining inference rules. Similarly, liveness of TRIO formulae could be analyzed, in connection or not with a possible use in rely/guarantee specifications.

An effort should be made in developing some form of high-level heuristics to guide and automate proofs of global properties in composite systems, at least in the most frequently occurring cases. These techniques could benefit from the analysis of the topology of the system and may aid the user in dividing a large proof and in building it.

Finally, it would be interesting to apply the proposed techniques to the specification and verification of a system larger than the simple ones discussed as examples in the thesis, possibly formalizing relevant parts of a realistic real-time system. Such a case study would constitute an interesting test bed for the applicability of the proposed methods and a step towards more realistic and industrial-size applications. It is currently in preparation.

APPENDICES

Appendix A

TRIO/PVS THEORIES

This appendix lists the PVS code of the auxiliary theories used to encode the modular extensions of TRIO (section A.1, described in chapter 4) and the rely/guarantee proof rule (section A.2, described in chapter 6).

A.1 Theory for modular TRIO extensions

```
TRIO_modular  [T: TYPE]
   : THEORY

BEGIN

   H1, H2: VAR T

   connect(H1, H2): boolean = (H1 = H2)

   % usage:
      % connections: AXIOM
      % connect(e1, e2) AND connect(e3, e4) AND ...
   %
   % or:
      % connection_1: AXIOM
      % connect(e1, e2)
      %
      % connection_2: AXIOM
      % connect(e3, e4)
      %
      % ...

END TRIO_modular
```

A.2 Theory for rely/guarantee reasoning

```
TRIO_relyguarantee  [N: nat] % N is the number of classes we are composing
   : THEORY

BEGIN
```

Appendix A (Continued)

```
IMPORTING trio_base, trio_lemmas

t: VAR Time
E, M: VAR TD_Fmla

%\rgarrowplus operator (this definition is better for rewrites)
>>=(E, M)(t): boolean = ( AlwP_e(E)(t)
                          IMPLIES (AlwP_i(M)(t) AND NowOn(M)(t)) )

%Initialized formula
Initialized?(E): boolean = Som( AlwP_e(E) )
Initialized_Fmla: TYPE+ = { E | Initialized?(E) }



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% RELY/GUARANTEE PROOF RULE FOR N COMPOSED CLASSES %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

rng: TYPE+ = {i: nat | i > 0 AND i <= N} % 1..N classes

IMPORTING trio_quantif[rng]

Rng_Fmla_Type: TYPE+ = [rng -> TD_Fmla]

P_i, Q_i: VAR Rng_Fmla_Type

j: VAR rng

>>=(P_i, Q_i): Rng_Fmla_Type = (LAMBDA j: (P_i(j) >>= Q_i(j)))

rwrt: FORMULA
Alw( (P_i >>= Q_i)(j) IFF (P_i(j) >>= Q_i(j)) )

rwrt_2: FORMULA
(P_i >>= Q_i)(j)(t) = (P_i(j) >>= Q_i(j))(t)

AUTO_REWRITE+ rwrt_2


E_i: VAR [rng -> Initialized_Fmla]
M_i: VAR Rng_Fmla_Type
E_g, M_g: VAR TD_Fmla


Rely_Guarantee_inference_rule: THEOREM
   (Alw( E_g AND FA(M_i)  IMPLIES  FA(E_i) )
```

```
      AND   Alw( FA(M_i)  IMPLIES  M_g )
      AND   Alw( FA(E_i >>= M_i) ) )
   IMPLIES
   Alw( E_g >>= M_g )


END TRIO_relyguarantee
```

Appendix B

PVS STRATEGIES FOR TRIO

This appendix lists the full LISP code for the PVS proof strategies described in chapter 6 and a pseudo-code description of what each strategy does, together with the required syntax for its invocation.

B.1 Proof strategies descriptions

syntax: (`open-fl` &OPTIONAL *fnum[*]*)
  repeat
    `open` given *fnum*
  until no more operators to open
  `flatten` given *fnum*

TABLE VI

`open-fl` proof strategy

Appendix B (Continued)

syntax: (`set-def-inst` *inst-str* &OPTIONAL *key[0]*)
  {*inst-str* is a list of instantiation values}
 Add mapping of *key* to the instantiation value *inst-str*
 Notify the user of the new mapping

TABLE VII

`set-def-inst` proof strategy

syntax: (`clear-def-inst`)
 Clear all global instantiations mappings

TABLE VIII

`clear-def-inst` proof strategy

syntax: (`def-inst` *fnums* &OPTIONAL *key[0]*)
 exprs ← mapping of *key*
 if $exprs = \emptyset$ **then** {*key* is not mapped to anything}
  error: notify the user
 else
  for each $fnum$ in *fnums* **do**
   `instantiate` $fnum$ with values *exprs*
  end for
 end if

TABLE IX

`def-inst` proof strategy

Appendix B (Continued)

syntax: (`lm-def-inst` *lemma* &OPTIONAL *key[0]*)
if *lemma* exists **then**
`lemma` *lemma* {introduce definition of lemma *lemma*}
`label` new formula with the full name of *lemma*
`def-inst` of new formula with mapping of *key*
else
error: notify the user
end if

TABLE X

`lm-def-inst` proof strategy

syntax: (`lm-def-use` *lemma* &OPTIONAL *key[0] time*)
`lm-def-inst` of *lemma* with *key*
if *time*$= \emptyset$ **then** {argument *time* not given}
`open` new formula and try heuristic instantiation of time
else
`open-inst` new formula at *time*
end if

TABLE XI

`lm-def-use` proof strategy

Appendix B (Continued)

syntax: (`connect` &OPTIONAL *dogrind? prefix key[0]*)

$lm_{name} \leftarrow$ concatenation of *prefix.* and "connections"
if lm_{name} exists **then**
  `lemma` lm_{name}
  `label` it with its full name
  `def-inst` of the new formula with *key*
else
  $idx \leftarrow 1$
  $lm_{name} \leftarrow$ concatenation of *prefix.* "connection_" and *idx*
  while lm_{name} exists **do**
    `lemma` lm_{name}
    `label` it with its full name
    `def-inst` of the new formula with *key*
    $idx \leftarrow idx + 1$
    $lm_{name} \leftarrow$ concatenation of *prefix.* "connection_" and *idx*
  end while
end if
if *dogrind?*$\neq \emptyset$ **then**
  `grind`
end if

TABLE XII

`connect` proof strategy

syntax: (`rg-use-definitions` &OPTIONAL *key[0]*)

`lm-def-use` of axiom "E_def" with *key*
`lm-def-use` of axiom "E_i_def" with *key*
`lm-def-use` of axiom "M_def" with *key*
`lm-def-use` of axiom "M_i_def" with *key*

TABLE XIII

`rg-use-definitions` proof strategy

Appendix B (Continued)

syntax: (`rg-i-case` *var* N &OPTIONAL *dogrind?[T]*)

```
if N ≤ 1 then
  if dogrind? = T then
    grind
  end if
else
  case var = N
  if dogrind? = T then
    grind
  end if
  recursive call of rg-i-case with arguments var N−1 dogrind?
end if
```

TABLE XIV

`rg-i-case` proof strategy

Appendix B (Continued)

B.2 Code of the proof strategies

B.2.1 General purpose strategies

```
(defstep open-fl (&optional (fnum *))
  (then (repeat
         (open fnum))
        (flatten fnum))
  "Does (open)*, then a flatten"
  "Opening and flattening formula ~a")
```

B.2.2 Strategies for class instantiations

```
;Hash table that maps keys to default instantiation sets
;Note that it remains the same through different proofs
;You can use both strings and numbers (since test #'equal is used)
(setf trio::def-inst-ht (make-hash-table :test #'equal) )

(setf trio::base-dummy-name "trio_dummy_name_")
(setf trio::cur-dummy-number 0)



(defhelper meaningful-skip ()
  (let ( (dummy_name (concatenate 'string trio::base-dummy-name
                              (prin1-to-string trio::cur-dummy-number)))
         (dummy_var (setf trio::cur-dummy-number
                          (+ 1 trio::cur-dummy-number))) )
    ;This is a dummy command so that the step is considered as actful
    ;and is recorded in the proof
    (then (name dummy_name "0")
          (delete -1))) ;Delete what the dummy command has done
  "Does nothing but makes the sequent be considered changed"
  "")



(defstep clear-def-inst ()
  (let ((dummy_var (setf trio::def-inst-ht (clrhash trio::def-inst-ht))))
    (comment "Default instantiation table cleared"))
  "Clears the table with all the default instantiations"
  "Clearing default instantiations table")



(defstep set-def-inst (inst-str &optional (key 0))
  (let ( (dummy_var (setf (gethash key trio::def-inst-ht) inst-str))
```

Appendix B (Continued)

```
        (cmsg (format nil "Default instantiation for key: ~a ~a set as: ~a"
                      key (if (equal 0 key) "(default)" "") inst-str)) )
    (then (meaningful-skip)
          (skip-msg cmsg)) )
  "Sets the default instantiations for class parametric lemmas.
   To be used in connection with def-inst"
  "Default instantiation set")


;This is just for recursive calls of def-inst strategy
(defhelper def-inst-aux (fnums exprs)
  (let ( (fnum (car fnums))
         (other-fnums (cdr fnums)) )
    (if (null fnum)
        (skip)
      (then (instantiate fnum exprs)
            (def-inst-aux other-fnums exprs))) )
  "Recursively does instantiation for each element of list of fnums"
  "")



(defstep def-inst (fnums &optional (key 0))
  (let ( (exprs (gethash key trio::def-inst-ht))
         (smsg (format nil
                      "There are no instantiation values for key: ~a" key))
         (list-fnums (if (listp fnums) fnums (list fnums))) )
    (if (null exprs)
        (skip-msg smsg)
      (def-inst-aux list-fnums exprs)) )
  "Instantiates all the fnums with default instantiation values
   stored under given key"
  "Providing default instantiations")



(defstep lm-def-inst (lemma &optional (key 0))
  (try (then (lemma lemma) (label lemma -1))
       (def-inst -1 key)
       (skip))
  "Introduces lemma and instantiates it with default instantiation values"
  "Instantiating lemma ~a with default instantiation values")



(defstep lm-def-use (lemma &optional (key 0) time)
  (try (lm-def-inst lemma key)
```

Appendix B (Continued)

```
      (try (if (null time) (open-inst -1) (open-inst -1 time))
           (open-fl -1)
           (skip))
      (skip))
  "Introduces lemma, instantiates it with default instantiation values,
   open and instantiates it at time, then does open-fl"
  "Introducing lemma ~a with default values and opening it")
```

B.2.3 Strategies for use of connections

```
(defstep connect (&optional dogrind? prefix (key 0))
   ;dogrind = nil not to do grind after connect,
   ;          any other value (e.g. t) to do it
   ;prefix is the name of the class where the connection are taken
   ;       (can be omitted if it's unabiguous)
   ;key is passed to def-inst
  (let ( (lm-name (concatenate 'string prefix (if (null prefix) "" ".")
                               "connections"))
         (dummy_var (setf trio::idx 0)) )
    (try (lemma lm-name)
         ;there's only one connection axiom: instantiate it and grind
         (try (then (label lm-name -1) (def-inst -1 key))
              (if (null dogrind?) (skip) (grind))
              (skip-msg "Default instantiation does not work
                         on the connection axioms"))
         ;need to instantiate all axioms connection_i
         (try
          (repeat (let ( (dummy_var_2 (setf trio::idx (+ 1 trio::idx)))
                         (lm-name (concatenate 'string prefix
                                               (if (null prefix) "" ".")
                                               "connection" "_"
                                               (prin1-to-string trio::idx))) )
                    (try (lemma lm-name)
                         (try (then (label lm-name -1) (def-inst -1 key))
                              (skip)
                              (skip-msg "Default instantiation does not work
                                         on the connection axioms"))
                         (skip))))
          (if (null dogrind?) (skip) (grind))
          (skip-msg "Couldn't find some of the connection axioms") ) ) )
    "Introduces and tries to instantiate according to default parameters
     the connection axioms for class <prefix>"
    "Using connection axioms")
```

Appendix B (Continued)

B.2.4 Strategies for rely/guarantee proofs

```
(defstep rg-use-definitions (&optional (key 0))
  (then (lm-def-use "E_def" key)
        (then (lm-def-use "E_i_def" key)
              (then (lm-def-use "M_def" key)
                    (lm-def-use "M_i_def" key))))
  "Introduces definitions for E, E_i, M and M_i assuming
   axioms of corresponding names exist"
  "Introducing definitions")



(defstep rg-i-case (var N &optional (dogrind? t))
    ;var = Skolem variable to switch on
    ;N = number of classes to be handled
    ;dogrind = t: tries to close cases with a grind,
    ;          any other value (e.g. nil) not to do it
  (if (<= N 1)
      (if dogrind? (grind) (skip))
    (spread@ (let ((rule (list 'case (concatenate 'string var
                                              " = " (prin1-to-string N)))))
               (quote rule))
             ( (if dogrind? (grind) (skip))
               (let ((nN (- N 1)))
                 (rg-i-case var nN dogrind?)) ) ))
  "Splits into N cases for each i (i.e. var) = 1, ..., N and grinds"
  "Splitting into subcases and grinding")
```

Appendix C

A FULL EXAMPLE OF AUTOMATED PROOF

This appendix lists the PVS theories translating the corresponding TRIO classes of the example in section 6.3 and illustrates with several details the proofs discussed in the same section, as they have been done in PVS.

C.1 System specification in PVS

```
echoer_rg  [instances: TYPE+]
  : THEORY

BEGIN

  IMPORTING trio_base, TRIO_modular, TRIO_relyguarantee[0]

  %% ITEMS

  input, output: [instances -> TD_Fmla] % in, out

  %% AXIOMS

  inst: VAR instances

  init: AXIOM
  AlwP_i(output(inst))(0)

  in_to_out: AXIOM
  Alw( input(inst) IMPLIES
          output(inst) AND Lasts_ii(output(inst), 1) )

END echoer_rg

two_echoers_rg  [instances: TYPE+]
  : THEORY

BEGIN
```

```
  IMPORTING trio_base, TRIO_modular, TRIO_relyguarantee[2]


  %% INSTANCES TYPES

  P1_Type: TYPE = { n: nat | n = 0} CONTAINING 0
  P2_Type: TYPE = { n: nat | n = 1} CONTAINING 1


  %% MODULES

  IMPORTING
     echoer_rg[P1_Type] AS P1,
     echoer_rg[P2_Type] AS P2


  %% CONNECTIONS

  p1: VAR P1_Type
  p2: VAR P2_Type

  connections: AXIOM
  connect(P1.output(p1), P2.input(p2)) AND
  connect(P2.output(p2), P1.input(p1))


  %% THEOREMS

  Rely_guarantee: THEOREM
  Alw(P1.output(p1) AND P2.output(p2))

END two_echoers_rg
```

C.2 Proof without strategies and rely/guarantee proof rule

C.2.1 Auxiliary lemmas for class `echoer_rg`

To follow the division of the proof we have discussed, we first of all introduce these two lemmas into the class `echoer_rg`.

Appendix C (Continued)

```
now_and_nexttime: LEMMA
Alw( input(inst) IMPLIES Futr(input(inst), 1) ) AND input(inst)(t)
         IMPLIES  (FORALL i: input(inst)(t + i))

alw_output: LEMMA
Alw( input(inst) IMPLIES Futr(input(inst), 1) ) AND input(inst)(t)
         IMPLIES  AlwF_i(output(inst))(t)
```

Note that `t` is a variable of type Time (i.e. real), while `i` is a variable of type natural.

Let us now consider the proof of these two lemmas. This proof is local to the class `echoer_rg`, so that it does not require any element which is not declared in this class.

Let us first prove `now_and_nexttime`, since the other one relies on this to be proven.

```
now_and_nexttime :

  |-------
{1}  FORALL (inst: instances, t: Time):
      Alw(input(inst) IMPLIES Futr(input(inst), 1))
        AND input(inst)(t)
              IMPLIES  (FORALL i: input(inst)(t + i))
```

After obvious introduction of Skolem variables and routinary formula manipulation, we get to the sequent:

```
now_and_nexttime :

{-1}  Alw(input(inst!1) IMPLIES Futr(input(inst!1), 1))
{-2}  input(inst!1)(t!1)
  |-------
{1}   FORALL i: input(inst!1)(t!1 + i)
```

We use induction on variable `i` to prove it.

The base case (`i = 0`) is trivial and is closed by an `assert`.

The inductive step:

Appendix C (Continued)

```
now_and_nexttime.2 :

[-1]  Alw(input(inst!1) IMPLIES Futr(input(inst!1), 1))
[-2]  input(inst!1)(t!1)
  |-------
{1}   FORALL j: input(inst!1)(t!1 + j)
                  IMPLIES input(inst!1)(t!1 + (j + 1))
```

is rather simple as well, since it only requires to manipulate the formulae `[-1]` and `[1]` with skolemizations and instantiations. So this proof is concluded easily.

Now, we consider the proof of the other lemma `alw_output`.

```
alw_output :

  |-------
{1}   FORALL (inst: instances, t: Time):
        Alw(input(inst) IMPLIES Futr(input(inst), 1))
          AND input(inst)(t)  IMPLIES   AlwF_i(output(inst))(t)
```

After the usual skolemizations and flattening of formulae, we get to the sequent:

```
alw_output :

[-1]  Alw(input(inst!1) IMPLIES Futr(input(inst!1), 1))
[-2]  input(inst!1)(t!1)
  |-------
{1}   output(inst!1)(nnt!1 + t!1)
```

Here we need to distinguish two cases: $nnt!1 = 0$ and $nnt!1 \neq 0$ so we issue the command `(case ‘‘nnt!1=0’’)` which yields two subgoals.

```
alw_output.1 :

{-1}  nnt!1 = 0
[-2]  Alw(input(inst!1) IMPLIES Futr(input(inst!1), 1))
[-3]  input(inst!1)(t!1)
```

```
  |-------
[1]   output(inst!1)(nnt!1 + t!1)

alw_output.2 :

[-1]  Alw(input(inst!1) IMPLIES Futr(input(inst!1), 1))
[-2]  input(inst!1)(t!1)
  |-------
{1}   nnt!1 = 0
[2]   output(inst!1)(nnt!1 + t!1)
```

Subgoal 1 is closed by a handful of commands: we need to introduce the axiom `in_to_out`, instantiate it at time `t!1` and finally launch a `grind`.

Subgoal 2 requires instead another case splitting: `(case integer?(nnt!1))`, so that we have the two following sequents, according to whether `nnt!1` is a natural number (since it is also nonnegative) or not.

```
alw_output.2.1 :

{-1}  integer?(nnt!1)
[-2]  Alw(input(inst!1) IMPLIES Futr(input(inst!1), 1))
[-3]  input(inst!1)(t!1)
  |-------
[1]   nnt!1 = 0
[2]   output(inst!1)(nnt!1 + t!1)

alw_output.2.2 :

[-1]  Alw(input(inst!1) IMPLIES Futr(input(inst!1), 1))
[-2]  input(inst!1)(t!1)
  |-------
{1}   integer?(nnt!1)
[2]   nnt!1 = 0
[3]   output(inst!1)(nnt!1 + t!1)
```

Appendix C (Continued)

To elaborate subgoal 2.1 we need to introduce the lemma `now_and_nexttime` we have just proved. We instantiate it at time `t!1`, call a `ground` to apply propositional reasoning with its `IMPLIES` operator and get to the formula

```
{-1}  FORALL i: input(inst!1)(i + t!1)
```

which we instantiate with `nnt!1`. Now we introduce axiom `in_to_out` one more time, instantiating it at `t!1 + nnt!1` so that a `grind` can conclude that

```
output(inst!1)(nnt!1 + t!1)
```

thus closing the sequent.

Subgoal 2.2 also needs the use of lemma `now_and_nexttime` with a `ground` command, getting to:

```
alw_output.2.2 :

{-1}  FORALL i: input(inst!1)(i + t!1)
[-2]  Alw(input(inst!1) IMPLIES Futr(input(inst!1), 1))
[-3]  input(inst!1)(t!1)
  |-------
[1]   integer?(nnt!1)
[2]   nnt!1 = 0
[3]   output(inst!1)(nnt!1 + t!1)
```

Since `nnt!1` is not an integer in this sequent, we need to instantiate the universal quantifier in formula -1 with the biggest integer which is also smaller than `nnt!1`. This quantity is by definition the *floor* function, so we instantiate with the command `(inst -1 ‘‘floor(nnt!1)’’)`, knowing that the function *floor* is defined in the PVS Prelude[1]. The next thing to do is to introduce the axiom `in_to_out` and instantiate it at time `t!1 + floor(nnt!1)` thus getting:

[1]The PVS Prelude is a built-in library of PVS theories with many mathematical functions of common use, together with their properties, available to the PVS system.

```
alw_output.2.2 :

{-1}  (input(inst!1) IMPLIES output(inst!1) AND
                 Lasts_ii(output(inst!1), 1))
          (t!1 + floor(nnt!1))
[-2]  input(inst!1)(floor(nnt!1) + t!1)
[-3]  Alw(input(inst!1) IMPLIES Futr(input(inst!1), 1))
[-4]  input(inst!1)(t!1)
  |-------
[1]   integer?(nnt!1)
[2]   nnt!1 = 0
[3]   output(inst!1)(nnt!1 + t!1)
```

We now exploit the implication of formula `{-1}` with an `open` and a `ground`, so that we get:

```
alw_output.2.2 :

{-1}  (output(inst!1) AND Lasts_ii(output(inst!1), 1))
              (floor(nnt!1) + t!1)
[-2]  input(inst!1)(floor(nnt!1) + t!1)
[-3]  Alw(input(inst!1) IMPLIES Futr(input(inst!1), 1))
[-4]  input(inst!1)(t!1)
  |-------
[1]   integer?(nnt!1)
[2]   nnt!1 = 0
[3]   output(inst!1)(nnt!1 + t!1)
```

We are now interested in the `Lasts_ii` subformula. We open it and instantiate at time `nnt!1 - floor(nnt!1)` to get the formula:

```
{-2}  output(inst!1)(nnt!1 - floor(nnt!1) + (floor(nnt!1) + t!1))
```

which can be recognized as the goal by a `grind` command which closes the sequent 2.2 and therefore the whole proof.

C.2.2 Auxiliary lemma for class two_echoers_rg

Before getting to the global property, we still need to introduce and prove a lemma in the class `two_echoers_rg`. We name the lemma `rg_aux`.

Appendix C (Continued)

```
rg_aux: LEMMA
Alw( P1.input(p1) IMPLIES Futr(P1.input(p1), 1) )
    AND  Alw( P2.input(p2) IMPLIES Futr(P2.input(p2), 1) )
```

The meaning of the lemma is simple. It basically says that, for both class `P1` and `P2`, the first hypothesis of lemmas `now_and_nexttime` and `alw_output` holds. This is a global property, since proving it requires the application of the axioms of both classes `P1` and `P2`.

The sequent to be proven is the following.

```
rg_aux :

  |-------
{1}   FORALL (p1: P1_Type, p2: P2_Type):
        Alw(P1.input(p1) IMPLIES Futr(P1.input(p1), 1)) AND
         Alw(P2.input(p2) IMPLIES Futr(P2.input(p2), 1))
```

After skolemization of universally quantified variables, this can be split into two subgoals.

```
rg_aux.1 :

{-1}  P1.input(p1!1)(tt!1)
  |-------
{1}   Futr(P1.input(p1!1), 1)(tt!1)

rg_aux.2 :

{-1}  P2.input(p1!1)(tt!1)
  |-------
{1}   Futr(P2.input(p1!1), 1)(tt!1)
```

As we can immediately understand, the proofs of the two subgoals are the same in their structures, except that in the second one we must use axioms and items of P2 whenever in the first one we used axioms of P1 and vice-versa. Being the two proofs so similar, we only analyze the one for `rg_aux.1`. Once that is built, we can modify its listing directly in PVS and redo the commands, adequately modified, to close the second part as well.

Appendix C (Continued)

The first thing to do is to introduce the axiom `in_to_out` for class `P1`, label it to remind it is from class `P1` and instantiate it at time `tt!1`. This produces the sequent:

```
{-1,(P1.in_to_out)}
      (input(p1!1) IMPLIES output(p1!1)
            AND Lasts_ii(output(p1!1), 1))(tt!1)
[-2]  P1.input(p1!1)(tt!1)
  |-------
[1]   Futr(P1.input(p1!1), 1)(tt!1)
```

After opening formula `{-1}`, a `ground` command recognizes the implication in the same formula and let us conclude that the consequent holds.

```
{-1,(P1.in_to_out)}
      (output(p1!1) AND Lasts_ii(output(p1!1), 1))(tt!1)
[-2]  P1.input(p1!1)(tt!1)
  |-------
[1]   Futr(P1.input(p1!1), 1)(tt!1)
```

Let us open formula `{-1}` and isolate the `Lasts` formula with a `flatten`, getting to the formula

```
{-2,(P1.in_to_out)}
      Lasts_ii(output(p1!1), 1)(tt!1)
```

We now expand the `Lasts` operator, which results in a universal quantification over the interval $[0, 1]$. We need to instantiate it at its upper bound, that is at 1. This leads us to the formula:

```
{-2,(P1.in_to_out)}
      output(p1!1)(1 + tt!1)
```

To use it, we have to introduce the axiom `in_to_out` for class `P2`, this time instantiating it at time `tt!1 + 1`.

```
{-1,(P2.in_to_out)}
      input(p2!1)(tt!1 + 1) IMPLIES
       (output(p2!1) AND Lasts_ii(output(p2!1), 1))(tt!1 + 1)
[-2,(P1.in_to_out)]
```

Appendix C (Continued)

```
      output(p1!1)(tt!1)
[-3,(P1.in_to_out)]
      output(p1!1)(1 + tt!1)
[-4]  P1.input(p1!1)(tt!1)
  |-------
[1]   Futr(P1.input(p1!1), 1)(tt!1)
```

On this sequent, we can issue a `ground` command to split the proof into two subgoals.

```
rg_aux.1.1 :

{-1,(P2.in_to_out)}
      output(p2!1)(1 + tt!1)
         AND Lasts_ii(output(p2!1), 1)(1 + tt!1)
[-2,(P1.in_to_out)]
      output(p1!1)(tt!1)
[-3,(P1.in_to_out)]
      output(p1!1)(1 + tt!1)
[-4]  P1.input(p1!1)(tt!1)
  |-------
[1]   Futr(P1.input(p1!1), 1)(tt!1)

rg_aux.1.2 :

[-1,(P1.in_to_out)]
      output(p1!1)(tt!1)
[-2,(P1.in_to_out)]
      output(p1!1)(1 + tt!1)
[-3]  P1.input(p1!1)(tt!1)
  |-------
{1,(P2.in_to_out)}
      input(p2!1)(1 + tt!1)
[2]   Futr(P1.input(p1!1), 1)(tt!1)
```

In sequent 1.1, we separate the `Lasts` from the other term in the `AND` formula. Moreover, we hide all the formula we no longer need to prove this sequent, getting to

```
[-1,(P2.in_to_out)]
      output(p2!1)(1 + tt!1)
  |-------
[1]   Futr(P1.input(p1!1), 1)(tt!1)
```

This is closed by means of the connection axiom, so we introduce it, instantiate for class parameters `p1!1` and `p2!1` and close the sequent with a `grind`.

Subgoal 1.2 is simple as well, since it reduces to:

```
[-1,(P1.in_to_out)]
      output(p1!1)(1 + tt!1)
  |-------
[1,(P2.in_to_out)]
      input(p2!1)(1 + tt!1)
```

once the useless formulae have been hidden. This can be closed with the introduction of the connections, too.

This concludes the proof of subgoal 1. As discussed, subgoal 2 is very similar and is not analyzed.

C.2.3 Proof of the global property of class `two_echoers_rg`

Now, we have all the ingredients to prove the global property of the composite class `two_echoers_rg`, that is:

```
Rely_guarantee: THEOREM
Alw(P1.output(p1) AND P2.output(p2))
```

As it is clear from all the previously seen cases, the proof of the theorem is divisible into two macro steps, where the second is structurally identical to the first one, except that it refers to class `P2` whenever the first step refers to class `P1` and vice-versa. Once again, we only discuss the proof for the first part, being the second easily extrapolable for any human reader (though not fully automatizable, since it requires the systematic changes to handle the different situation).

So, after initial routinary skolemizations and splitting into the two subgoals, the sequent we want to prove is:

Appendix C (Continued)

```
Rely_guarantee.1 :

  |-------
{1}   P1.output(p1!1)(tt!1)
```

where `tt!1` is a Skolem variable representing a generic time instant.

A case splitting is immediately needed, since the remainder of the proof is radically different whether we are considering time instants before or after 0. So, we issue the command `(case ''tt!1<=0'')` and get the two subgoals:

```
Rely_guarantee.1.1 :

{-1}  tt!1 <= 0
  |-------
[1]   P1.output(p1!1)(tt!1)

Rely_guarantee.1.2 :

  |-------
{1}   tt!1 <= 0
[2]   P1.output(p1!1)(tt!1)
```

where in 1.2, the condition `tt!1 <= 0` among the consequents is equivalent to the negation of the same condition among the antecedents, that is $tt!1 > 0$.

The proof of 1.1 is simple since it relies entirely on the initialization axiom of class `P1`. Hence, we introduce it with `(lemma ''P1.init'')`, label it, open its universal time quantification and instantiate it at time `- tt!1`, so that we have the formula:

```
{-1,(P1.init)}
      output(p1!1)(--tt!1 + 0)
```

Appendix C (Continued)

It is simple to recognize this is the same as the current goal, so that a `grind` closes it, togeter with an additional TCC (Type Correctness Contraint) generated autonomously by PVS.

The proof of sequent 1.2 is basically based on the axiom `alw_output` for class `P1`, so first of all we introduce it, label it and instantiate its free variable `t` at time 0 and its free variable `inst` with Skolem variable `p1!1`

```
{-1,(P1.alw_output)}
      Alw(input(p1!1) IMPLIES Futr(input(p1!1), 1))
              AND input(p1!1)(0)
       IMPLIES AlwF_i(output(p1!1))(0)
  |-------
[1]   tt!1 <= 0
[2]   P1.output(p1!1)(tt!1)
```

In order to show to the prover that the antecedent of the implication in formula `{-1}` is true, we need the axiom `P2.init` and the lemma `rg_aux`. More precisely, the former formula proves the truth of the second term of the conjunction, that is `input(p1!1)(0)`; the latter proves instead that the implication in the first term of the conjunction is also true.

Thus, we first introduce the lemma `rg_aux` and instantiate its instantiation parameters with the Skolem variables `p1!1` and `p2!1`. We get:

```
{-1}  Alw(P1.input(p1!1) IMPLIES Futr(P1.input(p1!1), 1)) AND
       Alw(P2.input(p2!1) IMPLIES Futr(P2.input(p2!1), 1))
[-2,(P1.alw_output)]
      Alw(input(p1!1) IMPLIES Futr(input(p1!1), 1))
              AND input(p1!1)(0)
       IMPLIES AlwF_i(output(p1!1))(0)
  |-------
[1]   tt!1 <= 0
[2]   P1.output(p1!1)(tt!1)
```

Appendix C (Continued)

Since the second term of the conjunction in formula `{-1}` refers to class `P2`, it is not needed in this branch of the proof. So we `split` that formula and hide it.

Now, we introduce the axiom `P2.init` and instantiate it at time 0 and for paramter `p2!1`. The sequent is now:

```
{-1,(P2.init)}
      output(p2!1)(-0 + 0)
[-2]  Alw(P1.input(p1!1) IMPLIES Futr(P1.input(p1!1), 1))
[-3,(P1.alw_output)]
      Alw(input(p1!1) IMPLIES Futr(input(p1!1), 1))
              AND input(p1!1)(0)
       IMPLIES AlwF_i(output(p1!1))(0)
  |-------
[1]   tt!1 <= 0
[2]   P1.output(p1!1)(tt!1)
```

It is now time to issue a `ground` command to exploit the implication if formula `[-3]`. This leads to the two subgoals:

```
Rely_guarantee.1.2.1 :

[-1,(P1.alw_output)]
      AlwF_i(output(p1!1))(0)
  |-------
[1]   tt!1 <= 0
[2]   P1.output(p1!1)(tt!1)

Rely_guarantee.1.2.2 :

[-1,(P2.init)]
      output(p2!1)(0)
  |-------
[1,(P1.alw_output)]
      input(inst)(0)
[2]   tt!1 <= 0
```

where we have already hidden the unnecessary formulae with a `hide` command.

Subgoal 1.2.1 is simply closed by first expanding the definition of the `AlwF` operator, instantiating the resulting universal quantifier at time `tt!1` and finally calling the usual `grind` command.

Subgoal 1.2.2 requires instead the connection axiom. Introducing it with a `(lemma ‘‘connections’’)`, instantiating its free variables and calling a `grind` suffices to close the sequent.

This also concludes the whole proof of the global property.

C.3 Proof with strategies and rely/guarantee proof rule

C.3.1 Proof of the local property

The initial sequent is:

```
rely_guarantee :

  |-------
{1}   FORALL (inst: instances): Alw(input(inst) >>= output(inst))
```

As usual, we first of all manipulate it by opening and skolemizing its `Alw` operator. Moreover, while introducing Skolem variable `inst!1` we also set it as the default instantiation value.

```
rely_guarantee :

{-1}  AlwP_e(input(inst!1))(tt!1)
  |-------
{1}   (AlwP_i(output(inst!1))(tt!1) AND NowOn(output(inst!1))(tt!1))
```

Now, we `open`, `flatten` and `split` the formula in `{1}`, getting two subgoals.

```
rely_guarantee.1 :

[-1]  AlwP_e(input(inst!1))(tt!1)
  |-------
{1}   AlwP_i(output(inst!1))(tt!1)
```

Appendix C (Continued)

```
rely_guarantee.2 :

[-1]  AlwP_e(input(inst!1))(tt!1)
  |-------
{1}   NowOn(output(inst!1))(tt!1)
```

Let us consider the sequent 1 first. We open and skolemize the goal formula, thus introducing a new Skolem variable `nnt!1` indicating the instants of time in which we need to prove `output` holds. We need to distinguish two cases, whether `nnt!1` is greater than 0 or it is less or equal to it. The two resulting subgoals are expressed by the sequents:

```
rely_guarantee.1.1 :

{-1}  nnt!1 > 0
[-2]  AlwP_e(input(inst!1))(tt!1)
  |-------
[1]   output(inst!1)(-nnt!1 + tt!1)

rely_guarantee.1.2 :

[-1]  AlwP_e(input(inst!1))(tt!1)
  |-------
{1}   nnt!1 > 0
[2]   output(inst!1)(-nnt!1 + tt!1)
```

In 1.1 we expand the definition of the `AlwP_e` operator and instantiate it at time `nnt!1`. Now, we need to show to the prover that `input` at time `tt!1 - nnt!1` also implies `output` at the same time. So we introduce the axiom `in_to_out` at time `tt!1 - nnt!1` and `grind` to close the sequent.

Appendix C (Continued)

Subgoal 1.2 requires instead to instantiate the quantification of the `AlwP_e` operator at some time instant before `tt!1` but not before `tt!1 - 1`. We choose, for instance, to instantiate it at `tt!1 - 1/2` with the command `open-inst`. This produces the sequent:

```
{-1}  input(inst!1)(-(1 / 2) + tt!1)
  |-------
[1]   nnt!1 > 0
[2]   output(inst!1)(-nnt!1 + tt!1)
```

Now, we introduce the axiom `in_to_out` and instantiate it at time `tt!1 - 1/2`. A `ground` command recognizes that the following now holds:

```
{-1,(in_to_out)}
      (output(inst!1) AND Lasts_ii(output(inst!1), 1))
                                  (tt!1 - (1 / 2))
[-2]  input(inst!1)(-(1 / 2) + tt!1)
  |-------
[1]   nnt!1 > 0
[2]   output(inst!1)(-nnt!1 + tt!1)
```

We are only interested in the term of the conjunction with the `Lasts` operator. So we separate it into a new formula with an `open-fl` command. Then, we expand the definition of the `Lasts` and instantiate the universal quantifier with value `1/2`. Finally, a `grind` command solves the mathematical equalities needed to recognize that subgoal 1 can be closed.

Let us prove subgoal 2. Similarly to what done in the other branch of the proof, we first of all need to instantiate the `AlwP` operator at some time before `tt!1` and after `tt!1 - 1`. We choose `tt!1 - 1/2`.

```
{-1}  input(inst!1)(-(1 / 2) + tt!1)
  |-------
[1]   NowOn(output(inst!1))(tt!1)
```

Appendix C (Continued)

Now, we introduce the axiom `in_to_out` once more, still instantiating it a time `tt!1 - 1/2`.

After that, a `ground` command produces the new sequent:

```
{-1,(in_to_out)}
      (output(inst!1) AND Lasts_ii(output(inst!1), 1))
                                      (tt!1 - (1 / 2))
[-2]  input(inst!1)(-(1 / 2) + tt!1)
  |-------
[1]   NowOn(output(inst!1))(tt!1)
```

We separate the `Lasts` formula into its own formula with and `open-fl` command. After that, we expand the definition of the `NowOn` operator. We need to instantiate the resulting existential quantifier for any value smaller than 1/2. We choose, for instance, 1/3, and get the sequent:

```
[-1,(in_to_out)]
      output(inst!1)(tt!1 - (1 / 2))
[-2,(in_to_out)]
      Lasts_ii(output(inst!1), 1)(tt!1 - (1 / 2))
[-3]  input(inst!1)(-(1 / 2) + tt!1)
  |-------
{1}   Lasts_ee(output(inst!1), 1 / 3)(tt!1)
```

Now, a handful of commands can close the sequent. More precisely, we apply `open-skolem` to formula `{1}` and make explicit the type for the newly introduced Skolem variable `it!1`. Then, we instantiate the expanded `Lasts` operator for the value `it!1 + 1/2`. After that, a simple `grid` command closes the sequent, thus concluding the proof.

C.3.2 Proof of the global property of class `two_echoers_rg`

We are ready to build the proof of the global property of the composite class `two_echoers_rg` using the rely/guarantee proof rule directly in PVS. In section 6.2 we have shown a basic guideline in defining additional items and definitions to use proficiently the rely/guarantee

Appendix C (Continued)

proof rule in PVS. Adhering to those guidelines, we add the following definitions to the class `two_echoers_rg`.

```
E: TD_Fmla = TRUE

E_i: Rng_Fmla_Type[2]
E_i_def: AXIOM
(E_i(1) = P1.input(p1)) AND (E_i(2) = P2.input(p2))

M: TD_Fmla
M_def: AXIOM
M = (P1.output(p1) AND P2.output(p2))

M_i: Rng_Fmla_Type[2]
M_i_def: AXIOM
(M_i(1) = P1.output(p1)) AND (M_i(2) = P2.output(p2))
```

Note that we did not define an axiom for the global environment assumption `E` since it is trivial and can be defined directly when we introduce the item `E`.

Now that the additional definitions have been introduced, the whole proof of the global property can be done without need for additional lemmas, directly from the basic proof sequent:

```
Rely_guarantee :

  |-------
{1}   FORALL (p1: P1_Type, p2: P2_Type):
        Alw(P1.output(p1) AND P2.output(p2))
```

We will only use the lemma for the local properties, proved in the previous section, the connection axiom, the definitions of the global and local assumptions and guarantees and the initialization axioms of the subclasses.

After introducing Skolem variables to replace the universal quantification of the goal, we want to save these new variables so that successive instantiations of lemmas can be done automatically by the prover. Thus, we set the default instantiation with the command `set-def-inst` as `p1!1 p2!1`. Moreover, we set other instantiation defaults for the values `p1!1` and `p2!1` alone, mapping keys `1` and `2` respectively. After that we open and skolemize the universal quantification on time of formula `{1}`, thus introducing the time Skolem variable `tt!1`. Now we are ready to apply the rely/guarantee inference rule we have defined in theory `TRIO_relyguarantee`. The sequent becomes:

```
{-1}  FORALL (E_g: TD_Fmla, E_i: [rng[2] -> Initialized_Fmla[2]],
                M_g: TD_Fmla, M_i: Rng_Fmla_Type[2]):
        (Alw(E_g AND FA(M_i) IMPLIES FA(E_i)) AND
          Alw(FA(M_i) IMPLIES M_g) AND Alw(FA(E_i >>= M_i)))
         IMPLIES Alw(E_g >>= M_g)
  |-------
[1]   (P1.output(p1!1) AND P2.output(p2!1))(tt!1)
```

To use it we have to choose which formulae represent the `E_g`, `E_i`, `M_g` and `M_i`. Since we have introduced in the theory the adequate items just before beginning this proof, we can immediately provide the instantiation with the command `(inst -1 "E" "E_i" "M" "M_i")`. This causes the proof to be split into two parts. The first one is the main branch and represents the use of the rely/guarantee proof rule. The second one is instead a TCC (Type Correctness Constraint) requiring to prove that `E_i` is initialized; this is needed to guarantee that the rule is sound.

Let us first consider how we discharge the TCC, represented by the sequent:

```
Rely_guarantee.2 (TCC):

  |-------
{1}   FORALL (x1: rng[2]): Initialized?[2](E_i(x1))
```

Appendix C (Continued)

After skolemizing the universal quantification in `{1}` we introduce the axioms `P1.init` and `P2.init` with instantiation values corresponding to keys `1` and `2` respectively. After that, we introduce the definition of `E_i` and expand that of the `Initialized` predicate, so that the sequent becomes:

```
Rely_guarantee.2:

{-1,(E_i_def)}
      (E_i(1) = P1.input(p1!1)) AND (E_i(2) = P2.input(p2!1))
[-2,(P2.init)]
      AlwP_i(output(p2!1))(0)
[-3,(P1.init)]
      AlwP_i(output(p1!1))(0)
  |-------
[1]   Som(AlwP_e(E_i(x1!1)))
```

Now, we realize that the prover must be aware that $AlwP_i(A) \Rightarrow AlwP_e(A)$ for any formula A. Thus, we introduce the formula `AlwP_i2AlwP_e` from the basic TRIO lemmas available in TVS theories. We provide two different instantiation for this lemma: one for `E_i(1)` and the other for `E_i(2)`, both at time `0`. Now the sequent is:

```
Rely_guarantee.2 :

[-1]  AlwP_i(P1.output(p1!1))(0) =
       (AlwP_e(P1.output(p1!1))(0) AND P1.output(p1!1)(0))
{-2}  AlwP_i(P2.output(p2!1))(0) =
       (AlwP_e(P2.output(p2!1))(0) AND P2.output(p2!1)(0))
[-3,(E_i_def)]
      (E_i(1) = P1.input(p1!1)) AND (E_i(2) = P2.input(p2!1))
[-4,(P2.init)]
      AlwP_i(output(p2!1))(0)
[-5,(P1.init)]
      AlwP_i(output(p1!1))(0)
  |-------
[1]   AlwP_e(E_i(x1!1))(0)
```

Appendix C (Continued)

Finally, we introduce the information about the connections and use the command `rg-i-case` on variable `x1!1` to close this branch of the proof.

Now, back to the main branch. The first thing to do is to introduce the definitions of the `E_i`, `E`, `M_i` and `M` items, so that the prover can use them as rewriting rules whenever needed during the proof. To do that we use the command `rg-use-definitions`, so that the sequent becomes:

```
Rely_guarantee.1 :

{-1,(M_i_def)}
      (M_i(1) = P1.output(p1!1))
{-2,(M_i_def)}
      (M_i(2) = P2.output(p2!1))
[-3,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
{-4,(E_i_def)}
      (E_i(1) = P1.input(p1!1))
{-5,(E_i_def)}
      (E_i(2) = P2.input(p2!1))
[-6,(E_def)]
      (Alw(E AND FA(M_i) IMPLIES FA(E_i)) AND
        Alw(FA(M_i) IMPLIES M) AND Alw(FA(E_i >>= M_i)))
       IMPLIES Alw(E >>= M)
  |-------
[1]   (P1.output(p1!1) AND P2.output(p2!1))(tt!1)
```

In order to exploit the implication of the rely/guarantee proof rule, we call a `ground` command which yields four different subgoals.

```
Rely_guarantee.1.1 :

{-1,(E_def)}
      Alw(E >>= M)
[-2,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-3,(M_i_def)]
```

```
      (M_i(2) = P2.output(p2!1))
[-4,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-5,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-6,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
[1]   (P1.output(p1!1) AND P2.output(p2!1))(tt!1)

Rely_guarantee.1.2 :

[-1,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-2,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-3,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-4,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-5,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
{1,(E_def)}
      Alw(E AND FA(M_i) IMPLIES FA(E_i))
[2]   (P1.output(p1!1) AND P2.output(p2!1))(tt!1)

Rely_guarantee.1.3 :

[-1,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-2,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-3,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-4,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-5,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
{1,(E_def)}
      Alw(FA(M_i) IMPLIES M)
```

```
[2]   (P1.output(p1!1) AND P2.output(p2!1))(tt!1)

Rely_guarantee.1.4 :

[-1,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-2,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-3,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-4,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-5,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
{1,(E_def)}
      Alw(FA(E_i >>= M_i))
[2]   (P1.output(p1!1) AND P2.output(p2!1))(tt!1)
```

Subgoal 1.1 requires to prove that the goal is subsumed by the global rely/guarantee formula $E \stackrel{+}{\rhd} M$, for the given `E` and `M`. This is trivial, and just requires to instantiate the rely/guarantee formula in `-1` at time `tt!1` and calling a `grind`. Subgoal 1.2 requires to prove the following hypothesis of the rely/guarantee proof rule we are using holds: $E \wedge \bigwedge_{i=1,\ldots,n} M_i \Rightarrow \bigwedge_{i=1,\ldots,n} E_i$. Subgoal 1.3 requires to prove the other hypothesis to the rely/guarantee proof rule: $E \wedge \bigwedge_{i=1,\ldots,n} M_i \Rightarrow M$. Finally, subgoal 1.4 requires to prove that the antecedent in the rely/guarantee proof rule statement holds, that is: $\bigwedge_{i=1,\ldots,n}(E_i \stackrel{+}{\rhd} M_i)$. Subgoals 1.2 through 1.4 are proved in the following paragraphs.

C.3.2.0.1 Proof of subgoal 1.2

The sequent to prove is, after hiding unnecessary formulae, introducing Skolem variables for `Alw` universal quantifications and flattening formulae:

Appendix C (Continued)

```
[-1,(E_def)]
      FA(M_i)(tt!2)
[-2,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-3,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-4,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-5,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-6,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
[1,(E_def)]
      FA(E_i)(tt!2)
```

In particular, we have hidden the formula `E(tt!2)` among the antecedents since it simplifies to `TRUE`.

We remind that the `FA` operators are universal quantifications over variable of type `rng`, that is the range of numbers $1, 2$ in this case, being two the subclasses we are composing. Thus, we introduce the Skolem variable `x!1` to indicate this parametrization with respect to `rng`, getting to the sequent:

```
{-1,(E_def)}
      FORALL (x: rng[2]): M_i(x)(tt!2)
[-2,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-3,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-4,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-5,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-6,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
```

Appendix C (Continued)

```
[1,(E_def)]
      E_i(x!1)(tt!2)
```

In order to avoid unnecessary splitting of the sequent, we make a copy of the formula `{-1}` so that we can instantiate it for $x = 1$ and its copy for $x = 2$. Moreover, we introduce the connection axiom with the command `connect` since it will be needed shortly.

```
{-1}  M_i(2)(tt!2)
[-2,(connections)]
      (P1.output(p1!1) = P2.input(p2!1)) AND
       (P2.output(p2!1) = P1.input(p1!1))
[-3,(E_def)]
      M_i(1)(tt!2)
[-4,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-5,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-6,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-7,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-8,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
[1,(E_def)]
      E_i(x!1)(tt!2)
```

Now we can realize that everything is complete to close the sequent. In fact, the `M_i` items are `output` items of the two classes, as described in their definitions. Moreover, the connections tell that each `output` is the `input` of the other class. Finally, the `input` items are by definition the `E_i` items. Thus, we just need to consider each case $1, 2$ separately with a command `(rg-i-case ''x!1'' 2)`. This completes the proof of subgoal 1.2.

C.3.2.0.2 Proof of subgoal 1.3

Subgoal 1.3 is rather simple to prove. First of all, we hide some unnecessary formulae, open and flatten other formulae and introduce a Skolem temporal variable `tt!2` to indicate the time of the `Alw` operator. So, the sequent we have to prove is:

```
{-1,(E_def)}
      FA(M_i)(tt!2)
[-2,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-3,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-4,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-5,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-6,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
[1,(E_def)]
      M(tt!2)
```

Similarly to what done in proof of subgoal 2, we expand the `FA` operator in formula `{-1}`, copy it and make two different instantiations for 1 and 2 (thus representing both classes `P1` and `P2`). We end up with the sequent:

```
{-1}  M_i(2)(tt!2)
[-2,(E_def)]
      M_i(1)(tt!2)
[-3,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-4,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-5,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-6,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-7,(E_i_def)]
```

```
      (E_i(2) = P2.input(p2!1))
  |-------
[1,(E_def)]
      M(tt!2)
```

Now a simple `grind` command recognizes that `M`, whose definition is in formula `[-6]`, holds, since both `M_1` and `M_2` hold. So the sequent is closed.

C.3.2.0.3 Proof of subgoal 1.4

The proof of subgoal 1.4 is rather simple as well, since it is basically reducible to the local theorem for class `echoer_rg` we have proved right above. Needless to say, we first of all introduce skolemizations for quantified time variables and hide some unnecessary formulae. We have the sequent:

```
[-1,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-2,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-3,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-4,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-5,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
{1,(E_def)}
      FA(E_i >>= M_i)(tt!2)
```

We now expand the definition of the `FA` operator with an `open` and replace the resulting universal quantification over a variable of type `rng` with a Skolem variable `x!1`. This leads to the sequent:

```
[-1,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
```

```
[-2,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-3,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-4,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-5,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
{1,(E_def)}
      (E_i(x!1) >>= M_i(x!1))(tt!2)
```

Now, we introduce the fundamental local lemmas of the subclasses. We adopt default instantiations and set the time at `tt!2`, that is we issue the command `lm-def-use` twice, for modules `P1` and `P2`. The sequent is now:

```
{-1,(P2.rely_guarantee)}
      (AlwP_e(input(p2!1))(tt!2) IMPLIES
      (AlwP_i(output(p2!1))(tt!2) AND NowOn(output(p2!1))(tt!2)))
[-2,(P1.rely_guarantee)]
      (AlwP_e(input(p1!1))(tt!2) IMPLIES
      (AlwP_i(output(p1!1))(tt!2) AND NowOn(output(p1!1))(tt!2)))
[-3,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-4,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-5,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-6,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-7,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
[1,(E_def)]
      (E_i(x!1) >>= M_i(x!1))(tt!2)
```

We are almost done: we still need to expand the definition of the `>>=` operator in the goal, which can be done with a `open`. Now, the sequent:

Appendix C (Continued)

```
[-1,(P2.rely_guarantee)]
      (input(p2!1)(tt!2) IMPLIES output(p2!1)(tt!2))
[-2,(P2.rely_guarantee)]
      (AlwP_e(input(p2!1))(tt!2) IMPLIES
      (AlwP_i(output(p2!1))(tt!2) AND NowOn(output(p2!1))(tt!2)))
[-3,(P1.rely_guarantee)]
      (input(p1!1)(tt!2) IMPLIES output(p1!1)(tt!2))
[-4,(P1.rely_guarantee)]
      (AlwP_e(input(p1!1))(tt!2) IMPLIES
      (AlwP_i(output(p1!1))(tt!2) AND NowOn(output(p1!1))(tt!2)))
[-5,(M_i_def)]
      (M_i(1) = P1.output(p1!1))
[-6,(M_i_def)]
      (M_i(2) = P2.output(p2!1))
[-7,(M_def)]
      M = (P1.output(p1!1) AND P2.output(p2!1))
[-8,(E_i_def)]
      (E_i(1) = P1.input(p1!1))
[-9,(E_i_def)]
      (E_i(2) = P2.input(p2!1))
  |-------
{1,(E_def)}
      (AlwP_e(E_i(x!1))(tt!2) IMPLIES
        (AlwP_i(M_i(x!1))(tt!2) AND NowOn(M_i(x!1))(tt!2)))
```

can be closed by issuing the command `rg-i-case` on the variable `x!1`. This triggers a very long sequence of rewritings which finally ends the whole proof.

CITED LITERATURE

1. Abadi, M. and Lamport, L.: The existence of refinement mappings. Theoretical Computer Science, 82(2):253–284, May 1991.

2. Abadi, M. and Lamport, L.: Composing specifications. ACM Transactions on Programming Languages and Systems, 15(1):73–132, Jan 1993.

3. Abadi, M. and Lamport, L.: An old-fashioned recipe for real-time. ACM Transactions on Programming Languages and Systems, 5(16):1543–1571, September 1994.

4. Abadi, M. and Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems, 17(3):507–535, May 1995.

5. Abadi, M. and Merz, S.: An abstract account of composition. Mathematical Foundations of Computer Science, 1995.

6. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996.

7. Alpern, B. and Schneider, F.: Defining liveness. Information Processing Letters, 21(4):181–185, October 1985.

8. Bjørner, N., Browne, A., and Manna, Z.: Automatic generation of invariants and intermediate assertions. Theoretical Computer Science, 173(1):49–87, February 1997.

9. Bjørner, N., Manna, Z., Sipma, H., and Uribe, T. E.: Deductive verification of real-time systems using STeP. Theoretical Computer Science, 253(1):27–60, 2001.

10. Bjørner, N., Browne, A., Chang, E., Colón, M., Kapur, A., Manna, Z., Sipma, H., and Uribe, T.: STeP: the Stanford Temporal Prover, user's manual. Technical report, Computer Science Department, Stanford University, November 1995.

11. Bjørner, N., Browne, A., Chang, E., Colón, M., Kapur, A., Manna, Z., Sipma, H., and Uribe, T.: STeP: deductive-algorithmic verification of reactive and real-time systems. In Proceeding of the 8th International Conference on Computer Aided

Verification, eds, R. Alur and T. Henzinger, volume 1102 of Lecture Notes in Computer Science, pages 415–418. Springer-Verlag, July 1996.

12. Chang, E., Manna, Z., and Pnueli, A.: Compositional verification of real-time systems. In Proceedings of the 9th IEEE Symposium on Logic in Computer Science, pages 458–465. IEEE Computer Society Press, 1994.

13. Chaochen, Z., Hoare, C., and Ravn, A.: A calculus of durations. Information Processing Letters, 40(5):269–276, 1991.

14. Cheng, A. M. K.: Real-Time Systems: Scheduling, Analysis, and Verification. John Wiley & Sons, August 2002.

15. Clarke, E., Emerson, E., and Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems, 8(2):244–263, 1986.

16. de Roever, W.-P.: The need for compositional proof systems: a survey. Lecture Notes in Computer Science, 1536:1–22, 1998.

17. Dijkstra, E.: Structured programming. In Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee, eds, J. Buxton and B. Randell, pages 84–88. NATO Science Committee, 1969.

18. Dijkstra, E.: A discipline of programming. Prentice Hall, 1976.

19. Duri, S., Buy, U., Devarapalli, R., and Shatz, S.: Using state space reduction methods for deadlock analysis in Ada tasking. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), pages 51–60, New York, 1993. ACM.

20. Fifi, A.: Un sistema per la conduzione di prove formali in TRIO basato su una codifica in PVS e un'interfaccia astratta. Laurea degree thesis, 1998. Politecnico di Milano.

21. Finkbeiner, B., Manna, Z., and Sipma, H. B.: Deductive verification of modular systems. Lecture Notes in Computer Science, 1536:239–275, 1998.

22. Floyd, R.: Assigning meanings to programs. In Proceedings of the AMS Symposium on Applied Mathematics, volume 19, pages 19–31, Providence, R.I., 1967. American Mathematical Society.

23. Frege, G.: Gedankengefüge, Beiträge zur Philosophie des Deutschen Idealismus, volume Band III, pages 36–51. 1923. Translation: Compound Thoughts, in P. Geach & N. Black (eds.), Logical Investigations, Blackwells, Oxford, 1977.

24. Gargantini, A. and Morzenti, A.: Automated deductive requirement analysis of critical systems. ACM Transactions on Software Engineering and Methodology (TOSEM), 10(3):255–307, July 2001.

25. Ghezzi, C., Mandrioli, D., and Morzenti, A.: TRIO: A logic language for executable specifications of real-time systems. The Journal of Systems and Software, 12(2):107–123, May 1990.

26. Grumberg, O. and Long, D. E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems, 16(3):843–871, May 1994.

27. eds, C. Heitmeyer and D. Mandrioli Formal Methods for Real-Time Computing. Trends in Software. John Wiley & Sons, 1996.

28. Hoare, C.: An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–580,583, 1969.

29. Holzmann, G. J.: The model checker SPIN. IEEE Transactions on Software Engineering, 23(5):279–295, 1997.

30. Hooman, J.: Compositional verification of real-time applications. In Compositionality: The Significant Difference (Revised lectures from International Symposium COMPOS'97), eds, W.-P. de Roever, H. Langmaack, and A. Pnueli, volume 1536, pages 276–300, Bad Malente, Germany, 1997. Springer-Verlag.

31. Jones, C.: Development methods for computer programs including a notion of interference. Doctoral dissertation, Oxford University Computing Laboratory, 1981.

32. Jones, C.: Tentative steps towards a development method for interfering programs. ACM Transactions on Programming Languages and Systems, 5(4):596–619, 1983.

33. Juan, E. Y. and Tsai, J. J.: Compositional Verification of Concurrent and Real-Time Systems. Boston, MA, Kluwer Academic Publishers, 2002.

34. Juan, E. Y., Tsai, J. J., and Murata, T.: Compositional verification of concurrent systems using Petri-nets-based condensation rules. ACM Transactions on Programming Languages and Systems, 20(5):917–979, 1998.

35. Kaufmann, M., Manolios, P., and Moore, J. S.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, June 2000.

36. Lamport, L.: The temporal logic of actions. ACM Transactions on Programming Languages and Systems, 16(3):872–923, 1994.

37. Lamport, L.: Composition: A way to make proofs harder. Lecture Notes in Computer Science, 1536:402–423, 1998.

38. Lichtenstein, O. and Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In Proceeding of the Twelfth Annual ACM Symposium on Principles of Programming Languages, January 1985.

39. Liu, J. W. S.: Real-Time Systems. Prentice Hall, 2000.

40. Manna, Z. and Pnueli, A.: Temporal verification of reactive systems: safety. New York, Springer-Verlag, 1995.

41. Manna, Z. and Pnueli, A.: Clocked transition systems. Technical report, Computer Science Department, Stanford University, April 1996.

42. McMillan, K. L.: Symbolic model checking - an approach to the state explosion problem. Doctoral dissertation, Carnegie Mellon University, 1992.

43. Misra, J. and Chandy, K.: Proofs of network of processes. IEEE Transactions on Software Engineering, 7(7):417–426, 1981.

44. Morasca, S., Morzenti, A., and San Pietro, P.: A tool for automated system analysis based on modular specifications. In The Thirteenth IEEE Conference on Automated Software Engineering (ASE'98), pages 2–11, Honolulu, Hawaii, USA, 1998. IEEE.

45. Morzenti, A. and San Pietro, P.: Object-oriented logical specification of time-critical systems. ACM Transactions on Software Engineering and Methodology, 3(1):56–98, January 1994.

46. Namjoshi, K. S. and Trefler, R. J.: On the completeness of compositional reasoning. In Proceedings of the 12th Int. Conference on Computer Aided Verification (CAV2000), number 1855, pages 139–153. Springer-Verlag, 2000.

47. Olderog, E.-R. and Dierks, H.: Decomposing real-time specifications. Lecture Notes in Computer Science, 1536:465–489, 1998.

48. Owre, S., Rushby, J. M., and Shankar, N.: PVS: A Prototype Verification System. In Proceedings of the 11th International Conference on Automated Deduction (CADE-11), ed. D. Kapur, volume 607 of Lecture Notes in Computer Science, pages 748–752, Saratoga Springs, NY, June 1992. Springer-Verlag. Also available online at `http://pvs.csl.sri.com/`.

49. Owre, S., Shankar, N., Rushby, J., and Stringer-Calvert, D.: PVS language reference. Computer Science Laboratory, SRI International, Menlo Park, CA. Also available online at `http://pvs.csl.sri.com/`.

50. Owre, S., Shankar, N., Rushby, J., and Stringer-Calvert, D.: PVS prover guide. Computer Science Laboratory, SRI International, Menlo Park, CA. Also available online at `http://pvs.csl.sri.com/`.

51. Owre, S., Shankar, N., Rushby, J., and Stringer-Calvert, D.: PVS system guide. Computer Science Laboratory, SRI International, Menlo Park, CA. Also available online at `http://pvs.csl.sri.com/`.

52. Raymond, E. S.: The jargon file. `http://www.catb.org/~esr/jargon/`. "bondage-and-discipline language" entry.

53. ed. E. S. Raymond The new hacker's dictionary, page 84. MIT press, third edition, 1996.

54. San Pietro, P., Morzenti, A., and Morasca, S.: Generation of execution sequences for modular time critical systems. IEEE Transactions on Software Engineering, 26(2):128–149, 2000.

55. Shankar, N.: Lazy compositional verification. Lecture Notes in Computer Science, 1536:541–564, 1998.

56. Shatz, S., Tu, S., Murata, T., and Duri, S.: An application of Petri net reduction for Ada tasking deadlock analysis. IEEE Transactions on Parallel and Distributed Systems, 7(12):1307–1322, December 1996.

57. Sistla, A. P.: Safety, liveness and fairness in temporal logic. Formal Aspects in Computing, 6:495–511, 1994.

58. Spivey, J. M.: The Z Notation: A Reference Manual. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

59. Vardi, M. Y.: On the complexity of modular model checking. In Logic in Computer Science, pages 101–111, 1995.

60. Woodcock, J. C. P. and Davies, J.: Using Z: Specification, Proof and Refinement. Prentice Hall International Series in Computer Science, 1996.

VITA

Personal data

- Full name: Carlo Alberto Furia
- Place and date of birth: Varese, Italy. September 16th, 1979
- Nationality: Italian
- Address: via Grandi 8, 21040 Venegono Inferiore (VA), Italy
- E-mail address(es): `cfuria1@uic.edu` `kurio@inwind.it` `caf@greatdante.net`

Undergraduate and graduate studies.

- Bachelor's degree: Politecnico di Milano, 2001.
- Master's degree: University of Illinois at Chicago, 2003, under the advising of Professor Dino Mandrioli.
- Laurea degree: Politecnico di Milano, 2003, under the advising of Professor Dino Mandrioli.

Publications.

- Furia, C.A.: Compositional Proofs for Real-Time Modular Systems. Master's thesis, University of Illinois at Chicago, Chicago, IL, 2003.
- Papers from the Master's thesis are currently in preparation

Languages

- Written and spoken Italian.
- Written and spoken English.