Università di Pisa

Dipartimento di Informatica

Technical Report : TR-99-10

A Refinement Calculus for Tuple Spaces

L. Semini, C. Montangero

March 15, 1997

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

A Refinement Calculus for Tuple Spaces

Laura Semini Carlo Montangero

Dipartimento di Informatica
Università di Pisa

Abstract

It is fairly accepted that the realization of complex systems must be accomplished step by step from the initial specification, through a sequence of intermediate phases, to the final program. These development steps, linking a preliminary version, or description, of the program to a more detailed one, are usually called *refinement* steps, while the intermediate stages of a refinement process are called *levels of abstraction*.

A *refinement calculus* is a means to support this modus operandi in program development, allowing to link different levels of abstraction: it introduces a precise relation between intermediate descriptions, and the rules to check whether the relation is satisfied.

Tuple space languages are concurrent languages, that foster the definition of autonomous entities of computation (the processes), and offer mechanisms for their synchronization and communication. In particular, they represent one of the most acknowledged models of *coordination*. Tuple space languages are based on the idea that a dynamic collection of tuples can act as shared state of concurrent processes, and play the role of coordination media among them.

To build a refinement calculus for tuple spaces, we address three points, in this paper:

1) We single out a specification language, a variation of first order temporal logic. Temporal relations between propositional formulae are not expressive enough to describe relations between tuple spaces, which are multisets of atoms. The specification language, called Oikos-tl, includes three new temporal operators that enhance the expressive power of the logic, permitting to directly link state transitions and state configurations. The semantics of the specification language is formally defined, and a set of useful properties for refinement are shown.

2) We introduce a reference language for tuple spaces, dubbed TuSpRel, and define its axiomatic and operational semantics. We need the former to derive properties, the latter to describe the allowed computations of a system. We relate these descriptions, and guarantee that using the axiomatic semantics we can derive properties, which are correct and complete with respect to the operational behaviour. The non-deterministic features of tuple space languages make this result new, and more complex than in other programming paradigms. One of the contributions of our work is the idea to derive weakest preconditions exploiting the *demonic strict* choice in non-deterministic selection. The transition system defining the operational semantics is based on the new notion of enabling precondition, which exploits the *angelic strict* choice.

3) To build the refinement calculus, we take a compositional approach. We first consider the basic statements of the language, and say under which conditions they satisfy a property, then compose these proofs to derive that a system refines a specification. Finally, in the refinement calculus definition, we extend to tuple space languages the ability to exploit logic formulae to specify the behaviour of unrefined modules: in the intermediate steps, a system is only partially written in the programming language, and the still unrefined features are described by logical formulae.

Contents

1 Introduction **3**
- 1.1 Framework . . . 3
- 1.2 Motivations . . . 4
- 1.3 State of the art . . . 5
- 1.4 Structure of the paper . . . 5

2 TuSpReL, a Reference Language **6**

3 Oikos-tl, a Specification Language **9**
- 3.1 State Formulae Logic . . . 10
- 3.2 Adding temporal operators . . . 12
- 3.3 Non standard Operators . . . 14
  - 3.3.1 Needs . . . 14
  - 3.3.2 After . . . 15
  - 3.3.3 Causes . . . 17

4 Semantics **18**
- 4.1 Motivations . . . 18
- 4.2 Preliminaries . . . 20
- 4.3 Weakest Precondition Semantics . . . 21
  - 4.3.1 Properties . . . 24
- 4.4 Towards an Operational Semantics: Enabling Preconditions . . . 24
  - 4.4.1 Enabling Precondition Semantics . . . 25
  - 4.4.2 Properties . . . 26
- 4.5 Operational Semantics: a Transition System . . . 26
- 4.6 Axiomatic vs operational semantics . . . 27

5 Refinement **28**
- 5.1 Refinement of safety properties . . . 29
- 5.2 Refinement of liveness properties . . . 31
- 5.3 Heterogeneity . . . 32

6 An Example: Eratosthenes' sieve **35**
- 6.1 Proof of the fourth refinement step: $E_3 \lhd E_4$ . . . 38
- 6.2 Proof of the sixth refinement step: $E_5 \lhd E_6$ . . . 38
- 6.3 Proof of the eighth refinement step: $E_7 \lhd E_8$ . . . 39

Bibliography **41**

A Appendix: Proofs from sections 3 and 4 **43**

1 Introduction

We discuss here the key features of tuple space languages, the motivations of our work, and the state of the art.

1.1 Framework

A coordination model is an interaction model between concurrent processes that communicate via a shared media, i.e., an interaction model for the components of a distributed system. The current large growth of local area networks is bringing the attention on distributed architectures, and coordination languages, providing a suitable communication model for distributed entities, have become popular in the last years [25]. One of the most acknowledged models of *coordination* is related to *tuple space languages*: these concurrent languages foster the definition of autonomous entities of computation (the processes), and offer mechanisms for their synchronization and communication. Tuple space languages are based on the idea that a dynamic collection of tuples can act as shared state of concurrent processes, and play the role of coordination media among them. The operations to read and change a tuple space represent the primitive statements of these languages. They exploit pattern matching or unification; program variables are logic variables (like in Prolog) instead of identifiers. As a consequence, processes access the state values by matching the state content, rather that exploiting variable addresses. This feature is known as *associative* access to data and strongly characterises the paradigm. Indeed, associative access is an aspect of declarativity.

The tuple space paradigm directly addresses the implementation of coordination policies: *separation* [19] of the computation and communication facets of a process, and *uncoupling* [18] between concurrent processes are the key features towards this aim.

Separation is achieved thanks to a suitable syntactic structuring of programs: process descriptions consist of a communication component, i.e. the read-write operations on the shared state, and a separate computation component, a sequential program. The program defining the computation component can be written in any sequential language; notably, many of them chose Prolog.

Uncoupling enhances modularity: no interaction between different processes must be explicitly described by the programmer. In particular, a process does not have to know the identity of the processes with which it cooperates, as it happens in the message passing paradigm, nor to deal with explicit synchronization and mutual exclusion constructs, as required in a shared variables paradigm. Communication via the tuple space is asynchronous, and needs no explicit addressing, thus fully satisfying the uncoupling requests. The processes involved in a computation can freely work both in parallel (along the time dimension) and in concurrency (along the space dimension), accessing the same store [18]. Actually, a tuple is a convenient way to communicate: the presence of a tuple in the tuple space represents a signal; moreover, tuples carry values.

Besides, tuple space languages have declarative facets. Declarativity is an advantage for the programmer, and we believe that this property is most useful when the programming goal is to build a concurrent application. In these cases, the declarative approach allows to describe events and reactions, while the procedural alternative requires to explicitly receive external inputs, test complex multiple conditions, and produce proper outputs.

Actually, the combination of tuple spaces with logic programming yields a paradigm that well integrates cooperation and declarativity. The expressive power of a tuple space language is enhanced by unification: process descriptions can include complex conditions on the tuple space, yet expressed in a very concise manner. The cooperation and the computation facets of a process (the interaction with the tuple space and the computation fired by the specified configurations of the tuple space, resp.) can be uniformly described by the same formalism.

The tuple space paradigm inspired the design of many programming languages, like Linda [18, 20], Swarm [48], LO [7], Γ [12, 13], Tao [47, 46, 42], manifold [9], SP [14], ESP [17], PoliS [24], Pâté [6], and *eta* [4, 5]. A focussed discussion of the characteristics of some of these languages can be found at the beginning of section 2

1.2 Motivations

Our target is the definition of a *logical, heterogeneous*, and *compositional* refinement calculus for tuple space languages. We fix the meaning of these attributes, and discuss why they are important for a refinement calculus. In the next section, we discuss in this perspective the recent works on refinement in tuple spaces, and outline the possibilites for improvement.

First the *logical* nature of a refinement is discussed, and compared with the opposed *semantic* one. In the logical approach we say that $P' \rhd P$ (P' refines P) if, given an appropriate set S of properties (logic formulae), $\forall f \in S$: P' satisfies f whenever P satisfies f. As an example a sequential sorting algorithm, satisfying the property that the output list is correctly sorted, is refined by a parallel algorithm satisfying the same property. In the semantic approach, on the contrary, we say that $P' \geq P$ if the semantics of P is an appropriate abstraction of the semantics of P'. As an example take the trace semantics of an algorithm A, and of a more detailed version of it, A', that adds some new information. We can say that the latter refines the former if we can find a suitable observation function that reduces A' semantics to be equivalent to that of A. The difference is methodological, in the first case results are based on a static verification of logical properties, while in the second case they are based on an analysis of the semantic domain.

A very general logical framework is provided by Hoare [31, 32], who claims that "programs are predicates". Any program P is identified with the strongest predicate describing its behaviour, and a calculus for deriving program predicates from basic statement predicates is proposed. The refinement relation then collapses to logical implication.

Morgan's approach [43, 44] is logical as well. It is based on the weakest precondition semantics, and the refinement relation is defined as follows: given two programs P and P', P' refines P if and only if for all formulae f: $wp(P,\, f) \;\rightarrow\; wp(P',\, f)$.

On the semantic side, Broy's approach is worth considering, and deals with the refinement of networks of interactive modules [15, 16]. A functional specification technique is used to supply the semantics of each module: a specification is a set of properties that the I/O behaviour must satisfy, the semantics is the set of I/O functions satisfying the specification. The refinement of a module specification P is a specification P' with $P' \rightarrow P$. Note that $P' \;\rightarrow\; P$ if and only if for every I/O function f: $P'(f) \;\rightarrow\; P(f)$.

We prefer the logical approach, because we can provide a general framework in which we can discuss a refinement relation between programs, as well as a satisfaction relation of a program with respect to a specification given in a logical language. On the contrary the semantic approach only allows to compare pairs of programs. Besides, the choice of focusing on the I/O behaviour, as made by Broy, is limitative, since it does not allow, for instance, to express properties of the computations, like safety or liveness, that are far-reaching when describing concurrent systems.

Let us now discuss what we mean with *heterogeneity*. This property deals with the intermediate steps of a step-wise derivation, the *unrefined programs*, which can either be written in a programming language (e.g., the language of the final program), or in a different non executable (e.g., specification) language. In the first case we have the advantage that we can execute and test the unrefined programs, while in the second case we are discharged from considering inessential, operational details. The heterogeneous approach belongs to the second case: each intermediate step is partially written in a programming language, and some unrefined features are described by logical formulae. In other words, we allow to exploit logic formulae to specify the behaviour of unrefined programs as, for instance, in [1]. In our opinion, this solution well meets the idea of step-wise refinement: the designer can, for instance, concentrate on a part of the program assuming that the others behave correctly.

Finally, we take into account *compositionality*. In the context of refinement, compositionality allows to derive a refinement relation between complex systems from a refinement relation between subcomponents. Going back to the example seen above, in Hoare's works [31, 32], the monotonicity results about statement compositions with respect to the refinement relation support modular (compositional) refinement. Broy's approach is compositional as well: composition operators (including parallel composition) allow to derive the behaviour of complex systems from the semantics of their modules, and, enjoying a monotonicity property with respect to the refinement relation, support a compositional refinement.

It is actually self-evident that a compositional approach is more desirable than a non compositional one, in particular when dealing with the development of systems built by a large number of different components, and we need not to motivate further our preference.

1.3 State of the art

In her doctoral thesis [22, 23], XiaoJun Chen presents a formalism for the verification of multiple tuple space systems, built by stepwise refinement, namely ESP systems [17]. An ESP system consists of two kinds of active components: a set of agents and a set of subsystems having the same structure as an ESP system. A subsystem is treated as a system of which we have only a black–box view. The semantics of an ESP system is given in terms of labeled transition systems. Chen's approach to refinement can be characterized as semantic: a program P' refines a program P if the semantics of P is an appropriate abstraction of the semantics of P'. We contend here that an approach based on logic is better for system development.

We find an example of logical approach to refinement in tuple spaces in the work on Swarm [26, 49]. Their derivation strategy starts with a specification of the program, which describes initial and final states, and some global properties that must hold during program execution. The initial specification is refined, producing a very detailed version that can be directly mapped into a program. Correctness of the derivation steps is proved by applying the *Swarm proof logic* [26], an assertional logic similar to that of Unity [21]. The building blocks of their proofs are assertions of the kind $\{p\}\, R\, \{q\}$, with R rewriting rule. However, they lack a formal derivation for these assertions. The difficulty mainly lies in the nondeterministic selection of the tuples made by the matching of the rule guard with the tuple space. The definition of a formal method to prove $\{p\}\, R\, \{q\}$, allowing this gap to be closed, represents one of the main results of this paper (see section 4).

1.4 Structure of the paper

In section 2 we define TuSpRel, a reference tuple space language and provide an example program. In TuSpRel, processes are reactive *agents* described in terms of *rules*, where a rule defines the interaction protocol with the tuple space: a guard defines conditions on the tuple space (firing conditions); a query starts a sequential inner computation in a given calculus; and a tell defines a write operation on the tuple space.

Our refinement calculus is based on a temporal logic oriented specification language, presented in section 3. In tuple spaces, a state is composed of ground atoms and a computation is a sequence of states: state formulae talk about the presence (or absence) of some atoms, general formulae, integrating state formulae and temporal operators, describe computations. In state formulae we capture the fact that states are *multisets* of atoms by introducing a non idempotent conjunction. We inherit Unity logic [21] and introduce three new temporal operators, suited to directly relate the *appearance* (instead of generic presence) of a tuple to other conditions of the current state of a computation. We refer to this specification language as Oikos-tl, since it has been inspired by the experience with previous developments in the Oikos project [3, 39].

In section 4 we define both an axiomatic and an operational semantics for our prototype language. The axiomatic semantics is in the weakest precondition style, and the transition system defining the operational semantics is based on the new notion of enabling precondition, which exploits the angelic choice in non-deterministic selection. We relate operational and weakest precondition semantics so that the properties derived using the axiomatic semantics are correct and complete with respect to the operational behaviour. This result overcomes the problems related to the basic computational statement, which is a non-deterministic rule rather than a simple assignment.

In section 5 we show how to refine Oikos-tl specifications into TuSpReL programs. We first deal with safety properties, showing when a rule refines a given property, and how to compose these proofs to derive the more general relation that a system refines a set of safety constraints. We then treat liveness. In the second part of the section, we introduce heterogeneous systems and propose a methodology of refinement.

Finally, in section 6 we provide an example of refinement, the problem addressed being Eratosthenes' sieve.

2 TuSpReL, a Reference Language

In almost all tuple space languages[1], processes are reactive *agents* described in terms of *rules*. We are interested, in this work, in the inspection of these languages. A rule defines the interaction protocol with the tuple space and can be sketched in the following way: a guard defines conditions on the tuple space (firing conditions); a query starts a sequential inner computation in a given calculus; and a tell defines a write operation on the tuple space. The inner computation is functional and affects the tuple space in terms of values assigned to the arguments of the tell. Linda is a notable exception: the processes have their own state and flow of control and exploit the tuple space only for coordination purposes. An atomic interaction with the tuple space involves only one tuple, i.e., each interaction consists in the read, consume, or write of a single tuple.

We fix here a prototype language based on tuple spaces and rules, as a common subset of the languages of this paradigm, that will act as reference language in the rest of the work. Formal syntax is provided in table 1. For the sake of reference, we call it TuSpReL, for Tuple Space Reference Language.

A tuple space consists of a multiset of ground atoms. A program is composed of set of rules. Optionally, it can include a Prolog program and a partial order between rules, expressing priorities. We assume the number of rules of a system to be invariant during system execution: no new rule is activated at run time. Rules are composed by a guard, a commit operator, an optional body, and a tell, and take the form:

GUARD | BODY. TELL

The GUARD is a sequence of *ask* and *consume*: the formers are queries on the contents of the tuple space, the latters indicate the set of atoms that must be removed from it. Moreover, the guard can contain some Prolog primitives, e.g. $X > Y, X \neq Y$. Ask, consume, and write operations are performed exploiting unification. An ask condition can take the form A as well as **not** A; in the first case it is satisfied if and only if the tuple space contains an atom B unifying with A, while in the second case the tuple space must not match any instance of A. The BODY is an (optional) Prolog atomic goal, evaluated with respect to the the Prolog program. TELL is a sequences of atoms and specifies a write operation. It can be partitioned in two parts: the first part (or the unique one if no partition exists) is executed in the case of empty BODY or successful evaluation of if. The second part is considered in the case of finite failure during BODY evaluation. We require that all the variables appearing in the TELL are instantiated when the tell operation is made, since we want only ground atoms in the tuple

[1] namely, Swarm, Γ, SP, ESP, PoliS, Pâté, *eta*, and Tao.

PROGRAM	::=	RULES BODY_DEFINITION PRIORITIES
RULES	::=	LABEL : RULE $\mid$ RULES RULES
RULE	::=	GUARD $\Big\vert$ POST % $\Big\vert$ is the commit operator
GUARD	::=	_ $\mid$ CONS GUARD $\mid$ ASK GUARD
CONS	::=	**cons** ATOM
ASK	::=	**ask** ATOM $\mid$ **ask not** ATOM $\mid$ **ask** TEST
TEST	::=	X = Y $\mid$ X $\neq$ Y $\mid$ ...
POST	::=	BODY TELL; TELL $\mid$ BODY TELL $\mid$ TELL
BODY	::=	**body** ATOM.
TELL	::=	_ $\mid$ **tell** ATOM $\mid$ TELL TELL
BODY_DEFINITION	::=	_ $\mid$ **with** *prolog program*
PRIORITIES	::=	_ $\mid$ **and** ORDER
ORDER	::=	LABEL $<$ LABEL $\mid$ ORDER ORDER
LABEL	::=	*ground term*
ATOM	::=	*prolog atom*
X, Y	::=	*terms*

Syntactic categories are capitalized. The symbols ::= and $\mid$ belong to the BNF notation, while ; . : <, and $\Big\vert$ belong to the language, as well as the Prolog primitives =, ==, $\neq$, etc. Keywords and primitives of the language, are in boldface. The former set includes: **with, ask, cons, body**, and **tell**; the latter includes **not**

A rule is composed of two parts separated by the commit operator "$\Big\vert$". The first part, GUARD, is the rule guard. The second, POST, is a conjunction of the body and the tell of the rule. They may be empty, however a rule with an empty guard is usually undesired, since it would continuously be executed. Both ASK and CONS are sequences of tagged atoms. The ask guard (ASK) may contain primitives derived from Prolog (TEST), to express relations between the terms appearing the guard. POST consists of a body and a tell. The body (BODY) is a Prolog atomic atom. TELL is a sequence of write declarations. It can be composed of a unique part or of two parts, separated by ";".

Table 1: TuSpReL syntax

R_1 :	**cons**	$max(M)$		
	ask	$primes(P)$	**ask**	$M < P$
$\mid$	**body**	$M' \; is \; M+1.$		
	tell	$max(M')$	**tell**	$n(M')$
R_2 :	**cons**	$n(X)$		
	ask	$Y < X$	**ask**	$X mod\, Y = 0$
$\mid$				
R_3 :	**ask not**	*done*		
$\mid$				
	tell	*done*		
and $R_1 > R_3 \;\; R_2 > R_3$				

Table 2: Example in the prototype language: Eratosthenes' Sieve

space. A necessary condition to satisfy this constraint is that the variables occurring in the TELL also appear in the body or in a positive literal of the GUARD. Actually, in some tuple space languages this constraint is not imposed. However, the semantic definition would be much more complex in this case.

We will define the semantics of our language in section 4. We provide here a very simple example to give an intuitive idea of rule behaviour. The rule:

$$\textbf{cons}\; d(\mathrm{X}) \quad \textbf{ask}\; \mathrm{X} > 0 \quad \mid \; \textbf{body}\;\; \mathrm{Y}\; is\; \mathrm{X}{+}1. \;\; \textbf{tell}\; d(\mathrm{Y})$$

looks for an atom matching $d(\mathrm{X})$ with $\mathrm{X} > 0$. If no such atom is in the tuple space, the rule is not executed. Otherwise, the rule can be fired, i.e. the commit operator " | " is taken. One of the matching atoms, say $d(c)$, is non-deterministically selected and consumed (removed from the tuple space). The body calculates $c+1$. Tell writes $d(c+1)$ in the tuple space. Guard evaluation is an atomic operation and is independent on the order in which *ask* and *consume* conditions appear[2].

In table 2, we show an example program in the prototype language. The problem addressed is Eratosthenes' Sieve: the program produces all the prime numbers minor than a given number P, provided that the tuples max(2) and n(2) are initially in the space. The solution is a non-trivial example of cooperation among a producer and a consumer. Rule R_1 acts as number generator, and generates orderly the numbers less than P; rule R_2 acts as sieve; rule R_3 states the end of the computation. The priority declaration $R_1 > R_3$ guarantees that the computation is not terminated before all the natural numbers minor than P have been generated; $R_2 > R_3$ guarantees that the computation is not ended if the sieve can detect and remove some more non prime numbers. Note that the sieve and the number generator can behave concurrently. To this purpose, we introduced max, playing the role of a counter, instead of having a generator simply asking for the presence of $n(\mathrm{X})$, with $\mathrm{X} < \mathrm{P}$, and the absence of $n(\mathrm{X}{+}1)$ to produce $n(\mathrm{X}{+}1)$. In fact, in this simplified solution, the generator would continue to produce the numbers that are removed by the sieve.

[2]However, the praxis is to list binding conditions first.

3 Oikos-tl, a Specification Language

Our refinement calculus is based on a temporal logic oriented specification language, Oikos-tl. We first discuss the main differences between Oikos-tl and the best known temporal logics, and then we supply syntax and semantics.

1) Non idempotent conjunction. In most tuple space languages, the states of a computation are multisets, hence neither propositional nor first order logic classical languages are well suited to describe their properties. Programming with multisets as primitive data structure is often useful, in particular when prototyping: it allows to model concisely and naturally the sharing of resources that can appear with multiple occurrences. Multiset union is not idempotent and we want to reflect this property in the logic framework: we integrate temporal operators and a non-idempotent connective $+$ that resembles the $\otimes$ (*times*) connective of linear logic [30]. We contend that the resulting logic is more appealing to software developers. Besides, it turns out that our non-idempotent conjunction is a useful operator also when specifying tuple space systems like Swarm systems [48], where the tuple space is a *set* of tuples. It is fairly common to need to constrain the computation in such a way that no two atoms with a same predicate p appear at the same time in the tuple space. To specify this constraint with the classical $\wedge$ we would have to say something like:

$$\Box \sim [(p(\mathrm{X}_1,\ldots,\mathrm{X}_n) \wedge p(\mathrm{Y}_1,\ldots,\mathrm{Y}_n)) \wedge (\mathrm{X}_1 \neq \mathrm{Y}_1 \vee \ldots \vee \mathrm{X}_n \neq \mathrm{Y}_n)]$$

while

$$\Box \sim (p(\mathrm{X}_1,\ldots,\mathrm{X}_n) + p(\mathrm{Y}_1,\ldots,\mathrm{Y}_n))$$

is all we have to say if we can use a non-idempotent conjunction.

2) No explicit quantification. Still addressing readability in specifications, we want to avoid explicit quantifications in the formulae, miming the simplicity of logic programming. Then, a formula is interpreted as universally or existentially quantified, as stated by the semantics given in section 3.2.

The assumption is that, in almost all cases, there is a natural quantification of the variables appearing in a formula. For instance, the intuitive meaning of $p(x) \rightarrow \Diamond q(x,y)$ is $\forall x\, [p(x) \rightarrow \Diamond \exists y\, q(x,y)]$.

3) New operators. We inherit Unity logic [21] and introduce three new temporal operators: AFTER, NEEDS, and CAUSES. Existing operators as, for instance, UNLESS, ENSURES, and LEADS-TO are sometimes too strong in the very first refinement steps. In addition, they are not suited to directly relate the *appearance* (instead of generic presence) of a tuple to other conditions of the current state of a computation. The new operators enhance the expressive power of temporal logic when describing reactive systems as the ones we are taking into consideration, leading to more readable and concise specifications. In particular, they allow us to specify straightforwardly conditions relating state changes (transitions) to state configurations or to further reactions, as loosely as it is often needed in the very first refinement steps. Most of the time we want to say something like: "if p appears, then ..." without making choices depending on the fact that p would remain true or not. For instance, Unity's $p \mapsto q$ is stronger than "if p appears, then q will be true" (see section 3.3).

4) Variable and predicate interpretation. In tuple spaces, a state is composed of ground atoms and a computation is a sequence of states ruled by reactions to the state contents. Hence, it is natural that a formula describing a state (*state formula*) talks about the presence (or absence) of some atoms, and that a formula describing a computation relates state formulae by temporal operators. For instance, $\Box(p(\mathrm{X}) \rightarrow \Diamond q(\mathrm{X}))$ says that every state containing an atom $p(\mathrm{X})$ is eventually followed by a state containing $q(\mathrm{X})$.

According to logic programming [36], we define the interpretation of a predicate p in a state s as the set of tuples of terms $t_1, \ldots, t_n$ such that the atom $p(t_1, \ldots, t_n)$ belongs to s. This set clearly changes during a computation and, consequently, the interpretation of a predicate can change from a state to the next. On the contrary, in the literature [34, 38, 37, 45], predicates have a fixed interpretation.

Moreover, in the literature, the set of variables appearing in a formula is split in two subsets: *local* and *global* variables (called also flexible and rigid, respectively, by Manna and Pnueli [38]). Global variables have a fixed interpretation in the states of a computation. Local variables, on the contrary, can be interpreted differently in different states. For instance, in

$$\Box \forall v \; [(x = v) \rightarrow \Diamond (x = 2v)]$$

x is a local variable, v a global one. In our proposal, we do not distinguish between local and global variables, all variables being considered as global. This is a consequence of the computational model. In tuple space languages there are no identifiers, while the need for local variables is tightly related to identifiers, whose values change during a computation.

In the next section we describe state formulae, in section 3.2 we extend state formulae with temporal operators, and, in section 3.3, we introduce the non standard temporal operators and some theorems for them.

3.1 State Formulae Logic

A state formula represents an assertion on a program state: it specifies conditions on the presence, or absence, of some tuples (atoms in our setting). To define them, we fix two distinguished sets of predicate symbols: *Pred* and *Test*. The first one represents the set of predicates appearing in the states of a computation, while the second contains predicates inherited from Prolog primitives ($\neq$, $=$, is , ...) [3].

The semantic definition for these formulae is given in two steps: we provide a weaker notion of validity, that can be given in an inductive way, then we base the semantics on it (definition 3.3).

Definition 3.1 (State formulae)

An atom $p(\bar{\text{X}})$ with $p \in Pred$ or an atom $t(\bar{\text{X}})$ with $t \in Test$ are positive *formulae.*
If A and B are positive formulae, then so is: $A + B$.
If A is a positive formula then $\sim A$ is a negative *formula.*
If A is a positive or negative formula, then it is a state *formula.*
If A and B are state formulae, then so is: $A + B$.

A state formula A has hence the structure[4]

$$p_1(\bar{\text{X}}) + \ldots + p_n(\bar{\text{Y}}) + t_1(\bar{\text{Z}}) + \ldots + t_m(\bar{\text{W}}) + \sim B_1 + \ldots + \sim B_l$$

We call pos(A) the set of variables occurring $p_1(\bar{\text{X}}), \ldots p_n(\bar{\text{Y}})$, and require the variables occurring in $t_1(\bar{\text{Z}}), \ldots t_m(\bar{\text{W}})$ also to occur in pos(A)[5]*,*

Example 3.2 *A couple of examples of state formulae follow. Other examples are in table 3.*

$$A = p(\text{X}) + q(\text{Y}) + \text{Y} > \text{X} + \sim (q(\text{Z}) + \text{Z} < \text{X}) \qquad pos(A) = \{\text{X}, \text{Y}\}$$

$$B = p(\text{X}) + \sim (q(\text{Y}) + q(\text{Z}) + \text{X} \; is \; \text{Y} - \text{Z}) \qquad pos(B) = \{\text{X}\}$$

[3] In the following of the section we will use the prefix notation for these predicates too.
[4] This is always possible as + is a commutative operator.
[5] Otherwise, we move the test within a negative subformula.

This intricate definition prevents more general formulae with nested negations, like, for instance $\sim (A+ \sim (A+B))$, from being considered: in that case, both the semantic definition and the decision algorithm, which permits to decide if a multiset is a model for a state formula [51], would be uselessly complex. Moreover, according to our experience, they are not very useful in the practice.

Interpretations for state formulae are tuple space states, i.e. multisets of ground atoms. Given an interpretation s and state formula A, we define $s \models \langle A,\ \vartheta\rangle$, to be read: A holds in s with substitution ϑ, or equivalently s is a model for A with success substitution ϑ. The label w stays for *weak*; the stronger notion of validity, that we require, constrains substitution ϑ to bind only the variables that occur in the positive sub-formula of A.

Definition 3.3 (Semantics) *We say that a state s models a formula A (with substitution ϑ), written $s \models \langle A, \vartheta\rangle$ if and only if:*

$$s \models_w \langle A, \vartheta\rangle \qquad \textit{and} \qquad \vartheta_{|\,pos(A)} = \vartheta$$

Where $\vartheta_{|\,B}$ is the restriction of ϑ to the variables occurring in B; $\uplus$ is multiset union:

$$\begin{array}{lll}
s \models_w \langle p(\bar{\mathrm{X}}), \vartheta\rangle & \textit{iff} \;\; p(\bar{\mathrm{X}})\vartheta \in s & p \in \mathit{Pred} \\
s \models_w \langle p(\bar{\mathrm{X}}), \vartheta\rangle & \textit{iff} \;\; p(\bar{\mathrm{X}})\vartheta = \mathit{true} & p \in \mathit{Test} \\
s \models_w \langle \sim (A), \vartheta\rangle & \textit{iff} \;\; \forall \vartheta' \;\; \textit{not} \;\; s \models_w \langle A, \vartheta' \circ \vartheta\rangle &
\end{array}$$

$$s \models_w \langle A{+}B,\ \vartheta\rangle \quad \textit{iff} \quad \left\{ \begin{array}{l} s \models_w \langle A, \vartheta\rangle, \quad s \models_w \langle B, \vartheta\rangle \\ \exists s_1\, s_2 :\ s_1 \uplus s_2 \subseteq s, \quad s_1 \models_w \langle A, \vartheta\rangle, \quad s_2 \models_w \langle B, \vartheta\rangle \end{array} \right.$$

In other words, $p(\bar{\mathrm{X}})$ holds in a state s with substitution ϑ if and only if s contains the (ground) atom $p(\bar{\mathrm{X}})\vartheta$, for $p \in \mathit{Pred}$, or the (ground) atom $p(\bar{\mathrm{X}})\vartheta$ is true in the usual Prolog semantics [52] if $p \in \mathit{Test}$. A state s is a w-model for $\sim (A)$ with substitution ϑ if and only if it does not exist any (grounding) substitution ϑ' such that s is a w-model for A with substitution $\vartheta' \circ \vartheta$.

Finally, the sum. State s w-models $A+B$ if and only if: it w-models A; it w-models B; it can be partitioned in two sub-states w-modelling A and B, respectively. The first two conditions are mandatory, otherwise $a + \sim a$ would have a w-model ($\{a\} = \{a\} \uplus \{\ \}$ with $\{a\}$ w-modelling a and $\{\ \}$ w-modelling $\sim a$) while we want it to be false. Moreover, the third condition cannot be skipped: consider the formula $a + a$, we want a to appear at least twice in a w-model for it, while the first two conditions alone would allow $\{a\}$ to be a w-model. Some examples follow. Some more examples are in table 3.

Example 3.4

Let a be a propositional atomic formula. Then $\{a\} \models \langle a+ \sim(a+a), \varepsilon\rangle$. In fact, both a and $\sim(a+a)$ hold in $\{a\}$. Moreover, $\{a\} = \{a\} \uplus \{\ \}$ with $\{\ \}$ model of $\sim(a+a)$.

$\{\ \}$ models every negative formula.

$\{p(a),\, p(b)\} \models \langle p(\mathrm{X}) + p(\mathrm{Y}) + \sim [p(\mathrm{Z}) + p(\mathrm{Z})],\ \vartheta\rangle$, *with* $\vartheta \in \{\{\mathrm{X}/a,\ \mathrm{Y}/b\},\ \{\mathrm{X}/b,\ \mathrm{Y}/a\}\}$

$\{p(1),\, p(3)\} \models \langle p(\mathrm{X}) + p(\mathrm{Y}) + \mathrm{X} > \mathrm{Y},\ \{\mathrm{X}/3,\ \mathrm{Y}/1\}\rangle$

The following example shows the difference between semantics and weak semantics:

$$\{p(1), p(5), q(3)\} \begin{array}{c} \models_w \\ \not\models \end{array} \langle p(\mathrm{X}) + \sim(q(\mathrm{Y}) + \mathrm{X} \geq \mathrm{Y}),\ (\mathrm{X}/5, \mathrm{Y}/4)\rangle$$

In fact substitution $(\mathrm{X}/5, \mathrm{Y}/4)$ *also binds a variable not appearing in the positive sub-formula.*

A	s	$pos(A)$	$\Sigma = \{\vartheta \mid s \models_w A\}$	$\Sigma' = \{\vartheta \mid s \models A\}$
$\sim p(\mathrm{X})$	$\{q(a)\}$	$\emptyset$	$\{\varepsilon\}$	$\{\varepsilon\}$
$\sim p(\mathrm{X})$	$\{p(a)\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\sim (p(\mathrm{X}) + p(\mathrm{X}))$	$\{p(a), p(b)\}$	$\emptyset$	$\{\varepsilon\}$	$\{\varepsilon\}$
$p(\mathrm{X})+ \sim q(\mathrm{Y})$	$\{p(a), q(b)\}$	$\{\mathrm{X}\}$	$\{(\mathrm{X}/a, \mathrm{Y}/a)\}$	$\emptyset$
$p(\mathrm{X})+ \sim q(\mathrm{X})$	$\{p(a), q(b)\}$	$\{\mathrm{X}\}$	$\{(\mathrm{X}/a)\}$	$\{(\mathrm{X}/a)\}$
$p(\mathrm{X})+ \sim (q(\mathrm{Y}) + \mathrm{X} \neq \mathrm{Y})$	$\{p(a), q(a)\}$	$\{\mathrm{X}\}$	$\{(\mathrm{X}/a)\}$	$\{(\mathrm{X}/a)\}$

Table 3: Success substitutions

We did not find a complete proof system for our state logic. This is not surprising, since it is a kind of dual to the problem of finding a model for linear logic. In particular, in our case, the substitution principle does not hold. In fact, let a and b be propositional atomic formulae, $\vdash$ any derivation relation of a sequent calculus sound with respect to the semantics given in definition 3.3, then we would have:

$a+ \sim (a+a) \not\vdash \sim a$

$a+ \sim (a+b) \vdash \sim b$

$a+ \sim (a+a+a) \not\vdash \sim (a+a)$

However, in [51] we propose an algorithm to decide if a multiset is a model for a state formula.

3.2 Adding temporal operators

We add temporal operators to relate state formulae. In this section, we introduce a subscript σ to distinguish between the operators we use (with implicit quantification) and normal temporal operators[6]. The relation between them is given in definition 3.6, and defines the way to make the quantification explicit. We inductively construct σ-formulae out of state formulae.

Definition 3.5 (Well formed σ-formulae) *If A is a state formula, then A is a σ*-wff. *If p and q are σ-wff, then so are: $\neg_\sigma p$, $p \rightarrow_\sigma q$, $\Box_\sigma p$, $\Diamond_\sigma p$, $\bigcirc_\sigma p$, $p \wedge_\sigma q$.*

In σ-formulae only some of the usual equivalences are true, because of the implicit quantification. For instance, $\Diamond_\sigma$ cannot be defined in terms of $\Box_\sigma$ and is thus introduced explicitly. The derived operators are:

$p \vee_\sigma q = \neg_\sigma(\neg_\sigma p \wedge_\sigma \neg_\sigma q)$

$p \mapsto_\sigma q = \Box_\sigma(p \rightarrow_\sigma \Diamond_\sigma q)$

$p \, \textsc{unless}_\sigma \, q = \Box_\sigma((p \wedge_\sigma \neg_\sigma q) \rightarrow_\sigma \bigcirc_\sigma(p \vee_\sigma q))$

$\textsc{stable}_\sigma \, p = p \, \textsc{unless}_\sigma \, false$

$p \, \textsc{until}_\sigma \, q = (p \, \textsc{unless}_\sigma \, q) \wedge_\sigma p \mapsto_\sigma q$

We define a function t on σ-formulae, which makes quantification over variables explicit and is needed to define σ-formulae semantics. In the definition, we abbreviate with $\exists\vartheta_p$ the sentence: "$\exists\vartheta$, ϑ is restricted to the variables occurring in p" (similar for $\forall$), while with $\exists\vartheta_{p_{\setminus ex}}$ we mean: "$\exists\vartheta$, ϑ is restricted to the variables occurring in p and not already bound by an external substitution". For instance, in $\forall\vartheta_{p\cap q}\,[p(x)\vartheta \rightarrow \Diamond\exists\vartheta'_{q_{\setminus ex}}\langle q(x,y)\vartheta, \vartheta'\rangle]$, ϑ is substitution on x, and ϑ' a substitution on y.

[6] In the next sections we will deal only with σ-formulae, and simplify the notation dropping label σ

Definition 3.6 (Translation function from σ- formulae to temporal logic formulae)

$$t(A) = \exists\vartheta_{A_{\setminus ex}} \langle A, \vartheta\rangle \qquad A\ state\ formula$$

$$t(\neg_\sigma\, p) = \forall\vartheta_p\, \neg\langle t(p), \vartheta\rangle$$

$$t(p \to_\sigma q) = \forall\vartheta_{p\cap q}\, [t(p\vartheta) \to t(q\vartheta)]$$

$$t(\Box_\sigma p) = \exists\vartheta_{p_{\setminus ex}}\, \Box\langle t(p), \vartheta\rangle$$

$$t(\Diamond_\sigma p) = \exists\vartheta_{p_{\setminus ex}}\, \Diamond\langle t(p), \vartheta\rangle \;=\; \Diamond\, \exists\vartheta_{p_{\setminus ex}}\, \langle t(p), \vartheta\rangle$$

$$t(\bigcirc_\sigma p) = \exists\vartheta_{p_{\setminus ex}}\, \bigcirc\langle t(p), \vartheta\rangle \;=\; \bigcirc\, \exists\vartheta_{p_{\setminus ex}} \langle t(p), \vartheta\rangle$$

$$t(p \wedge_\sigma q) = \exists\vartheta_{(p\cup q)_{\setminus ex}}\, [\langle t(p), \vartheta\rangle \;\wedge\; \langle t(q), \vartheta\rangle]$$

As an example of translation, we show that $\Box_\sigma p \;\neq\; \neg_\sigma\Diamond_\sigma\neg_\sigma p$: $\quad t(\Box_\sigma p) \;=\; \exists\vartheta_p\Box\langle p, \vartheta\rangle$ $\neq\; \Box\exists\vartheta_p\langle p, \vartheta\rangle \;=\; \neg\Diamond\forall\vartheta_p\neg\,\langle p, \vartheta\rangle \;=\; \neg_\sigma\Diamond_\sigma\neg_\sigma p$. The semantics of $\exists\vartheta\Box\langle p, \vartheta\rangle$ and $\Box\exists\vartheta\langle p, \vartheta\rangle$ will be given in definition 3.8. However, we can anticipate that $\exists\vartheta\Box\langle p, \vartheta\rangle \;\neq\; \Box\exists\vartheta\langle p, \vartheta\rangle$: as discussed in the introduction of the section, in our approach the interpretation of a predicate can change in the states of a computation. In other words, in σ-formulae the equivalence $\Box_\sigma \;\equiv\; \neg_\sigma\Diamond_\sigma\neg_\sigma$ does not hold. On the contrary, the *Barcan formula* $\forall\,\Box \equiv \Box\,\forall$ is a theorem with respect to our semantics. In [51], we show that, exploiting the translation function t, a set of interesting equivalences can be found also in the case of σ-formulae.

We introduce the notion of *normal form* for the translation of a σ-formula. Normal forms are needed to define the semantics of σ-formulae in an inductive way (see definition 3.8). Let A be a σ-formula, we say that $t(A)$ is in *normal form* if for any sub-formula $\langle p, \vartheta\rangle$ of $t(A)$, $p = q\vartheta'$ with q state formula.

For instance, $\exists\vartheta_p\langle p, \vartheta\rangle \;\vee\; \exists\vartheta_q\langle q, \vartheta\rangle$ is in normal form, $\exists\vartheta_p[\exists\vartheta'_{q_{\setminus ex}}\langle p \wedge q, \vartheta'\rangle, \vartheta]$ is not. However, since ϑ and ϑ' refer to different sets of variables (this is always the case by definition 3.6), we can pass the substitutions through the brackets and obtain the normal form: $\exists\vartheta_p\exists\vartheta'_{q_{\setminus ex}}\langle p \wedge q, \vartheta'\vartheta\rangle$.

Definition 3.7 (Models) *A model $\mathcal{M}$ is a pair $\langle S, V\rangle$ with S infinite Noetherian denumerable chain of states (multisets), and V valuation function mapping a state $s \in S$ in the set $\{\langle A, \vartheta\rangle \mid s \models \langle A, \vartheta\rangle\}$*

We say that $\mathcal{M} = \langle S, V\rangle$ satisfies p if and only $t(p)$ holds in $\mathcal{M}$, s_0, where $\mathcal{M}$, $s_i|\!\models t(p)$ is inductively defined as follows (F denotes a sub-formula). The definition is standard. We provide it, for the sake of completeness.

Definition 3.8 (Semantics)

$$\mathcal{M},\, s_i|\!\models \langle A, \vartheta\rangle \quad \textit{iff} \quad \langle A, \vartheta\rangle \in V(s_i) \;\; \textit{with } A \textit{ state formula, i.e., if } s_i \models \langle A, \vartheta\rangle$$

$$\mathcal{M},\, s_i|\!\models \exists\,(\forall)\,\vartheta F \quad \textit{iff} \quad \exists\,(\forall)\,\vartheta\,\mathcal{M},\, s_i|\!\models F$$

$$\mathcal{M},\, s_i|\!\models \neg F \quad \textit{iff} \quad \textit{not}\ \mathcal{M},\, s_i|\!\models F$$

$$\mathcal{M},\, s_i|\!\models (F_1 \to F_2) \quad \textit{iff} \quad \mathcal{M},\, s_i|\!\models F_1 \;\to \mathcal{M},\, s_i|\!\models F_2$$

$$\mathcal{M},\, s_i|\!\models \Box F \quad \textit{iff} \quad \forall\, k \geq i.\, \mathcal{M},\, s_k|\!\models F$$

$$\mathcal{M},\, s_i|\!\models \Diamond F \quad \textit{iff} \quad \exists\, k \geq i.\, \mathcal{M},\, s_k|\!\models F$$

$$\mathcal{M},\, s_i|\!\models \bigcirc F \quad \textit{iff} \quad \mathcal{M},\, s_{i+1}|\!\models F$$

$$\mathcal{M},\, s_i|\!\models F_1 \wedge (\vee)\, F_2 \quad \textit{iff} \quad \mathcal{M},\, s_i|\!\models F_1 \textit{ and } (\textit{or})\, \mathcal{M},\, s_i|\!\models F_2$$

Example 3.9 *Let $S = \{p(a)\}, \{p(a), p(b), q(a)\}, \{p(b)\}, \{p(b)\}, \{p(b)\}, \ldots$ be a computation, then $\langle S, V\rangle$ satisfies the formulae $\Diamond_\sigma\Box_\sigma\, p(\mathrm{X})$ and $p(\mathrm{X})\textsc{unless}_\sigma\, q(\mathrm{X})$, while it is does not satisfy $\Box_\sigma p(\mathrm{X})$, nor $\Box_\sigma\neg_\sigma(p(\mathrm{X}) + p(\mathrm{Y}))$*

The next proposition states that the semantics of $\sim p(\bar{\mathrm{x}})$, seen as σ-formula, is equivalent to the semantics of $\neg\, p(\bar{\mathrm{x}})$, where "$\sim$" is the negation symbol we used in state formulae, and "$\neg$" the negation symbol we introduced when extending state formulae with temporal operators (see, respectively, definition 3.1 and definition 3.5). This result allows us not to distinguish $\neg$ and $\sim$ in the following. The proof is in the appendix.

Proposition 3.10 $\sim p(\bar{\mathrm{x}}) \;\equiv\; \neg\, p(\bar{\mathrm{x}})$

3.3 Non standard Operators

We introduce three new temporal operators. They are a sort of *past* operators and are introduced to lead to more readable and concise specifications, when describing reactive systems as the ones we are taking into consideration. In particular, they allow us to specify conditions related to the appearance (instead of presence) of a tuple in the tuple space. In other words they allow us to directly relate state changes to state configurations or to further reactions, i.e., to naturally relate transitions to state configurations.

The operators we introduce are: NEEDS, AFTER and $\hookrightarrow$ (causes)[7]. The operator NEEDS permits to express a consistency condition by constraining the states in which a new property may become true: p NEEDS q means that, if p appears, then q must be true as well. If p is true initially, then q must be true in the initial state too. With p AFTER q we define *necessary* causes: p can become true only if q was true in the previous state. Finally, the property $p \hookrightarrow q$ defines *sufficient* causes: it requires a state satisfying the conclusion q to follow (not immediately, though) the state in which the premise p becomes true, or to hold sometimes if p is true in the initial state. Formally, let $\hat{\bigcirc}$ indicate the previous state [38], and have the following semantics:

$$\mathcal{M},\, s_0 \Vdash\!\models\; t(\hat{\bigcirc} p)$$

$$\mathcal{M},\, s_{i+1} \Vdash\!\models\; t(\hat{\bigcirc} p) \quad \text{iff} \quad \mathcal{M},\, s_i \Vdash\!\models\; t(p)$$

Then:

$$p \;\text{NEEDS}\; q \;\stackrel{def}{=}\; \text{INIT}\,(p \to q) \;\wedge\; \Box\,[\, p \to (\hat{\bigcirc} p \vee q)\,]$$

$$p \;\text{AFTER}\; q \;\stackrel{def}{=}\; \Box\,[\, p \to \hat{\bigcirc}(p \vee q)\,]$$

$$p \hookrightarrow q \;\stackrel{def}{=}\; p \;\text{NEEDS}\; (\Diamond\, q)$$

As an immediate consequence of these definitions, we have the equivalence

$$p \;\text{AFTER}\; q \;\equiv\; p \;\text{NEEDS}\; \hat{\bigcirc} q$$

In table 4 we provide a representation of correct behaviours with respect to formulae including these operators and Unity operators. In the following, we list some useful properties of the new operators (the proofs are in [51]).

3.3.1 Needs

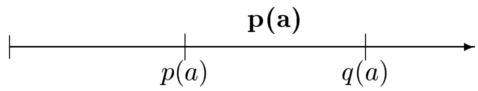
As an example of the use of NEEDS, imagine that somebody (P) wants to buy a software product S, and wants the most recent version V of S. With a natural interpretation of the predicates, this can be expressed as

$$has(P,V,S) \;\text{NEEDS}\; most\_recent(V,S)$$

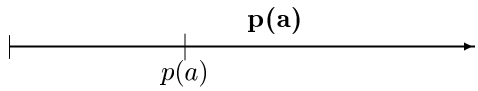
saying precisely that V is the most recent version of S when P gets it: remember that the left-hand side of NEEDS refers to the state *transition* that makes the operand true. Actually,

[7] Warning: from now on we drop label σ, since we use only σ-formulae.

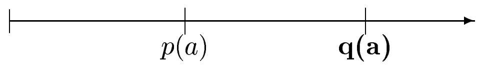
p(X) unless q(X)
once true, p(X) holds until q(X) becomes true



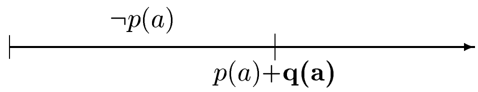
stable p(X)
once true, p(X) stays true



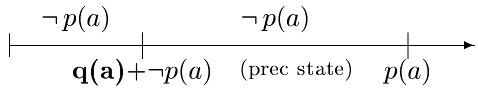
p(X) ↦ q(X)
q(X) eventually true after p(X) was true



p(X) needs q(X)
q(X) must be true when p(X) becomes true



p(X) after q(X)
q(X) must be true before p(X) becomes true



p(X) ↪ q(X)
q(X) eventually true after p(X) became true

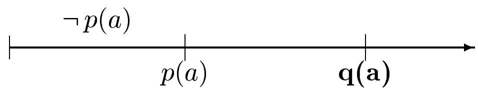


Table 4: Informal definition of temporal operators: predicates over the line refer to the time interval; predicates under the line refer to the time instant; predicates in boldface are those "caused" by the other conditions.

a new version of S can be released afterwards: the formula above permits P to continue and use the previous one, if he wants to. Such freedom would not be modelled by

$$has(P,V,S) \ \rightarrow \ most\_recent(V,S)$$

which forces P to keep updating the version he has. On the other side, the best we can do with UNLESS is

$$(\neg\, has(P,V,S)) \text{ UNLESS } most\_recent(V,S)$$

which, besides being definitely less readable, would allow P to get a version V of S (immediately) before its release, thus not describing precisely what we want to specify.

Operator NEEDS is reflexive (N1, table 5), distributes over the conjunction (N2), enjoys a closure property with respect to implication (N3), and entails two obligations, let p NEEDS q hold: if q is never satisfied, then p can never become true (N4), and if $\neg\, q$ NEEDS $\neg\, p$ holds as well, then q must be true whenever p is true (N5). Finally, rule N6 states a weak transitivity property.

3.3.2 After

To exemplify the use of AFTER, we consider flight reservations. A safety condition that we might want to describe is that a customer P gets a reservation on a flight F only if he made a request and if there were some seats free before the reservation is taken, i.e.

$$reserved(P,F) \text{ AFTER } (\, seats\_available(F) \ + \ request(P,F)\,)$$

Again, the use of operators $\rightarrow$ and $\hat{\bigcirc}$, as in

$$reserved(P,F) \ \rightarrow \hat{\bigcirc}\ (\, seats\_available(F) \ + \ request(P,F)\,)$$

Properties of NEEDS

$$p \ \textsc{needs}\ p \qquad \text{(reflexivity)} \qquad \mathbf{N1}$$

$$(p \ \textsc{needs}\ q) \ \wedge \ (p \ \textsc{needs}\ r) \ \equiv \ p \ \textsc{needs}\ (q \ \wedge \ r) \qquad (\wedge \text{ equivalence}) \qquad \mathbf{N2}$$

$$\frac{p \ \textsc{needs}\ q \qquad \Box\,(q \rightarrow r)}{p \ \textsc{needs}\ r} \qquad (\textsc{needs} \text{ - } \rightarrow) \qquad \mathbf{N3}$$

$$\frac{p \ \textsc{needs}\ q}{\Box(\neg q) \ \rightarrow \Box(\neg p)} \qquad (\textsc{needs} \text{ - obligation 1}) \qquad \mathbf{N4}$$

$$(p \ \textsc{needs}\ q) \ \wedge \ (\neg q \ \textsc{needs}\ \neg p) \ \equiv \ \Box\,(p \rightarrow q) \qquad (\textsc{needs} \text{ - obligation 2}) \qquad \mathbf{N5}$$

$$\frac{p \ \textsc{needs}\ q \qquad q \ \textsc{needs}\ r \qquad r \ \textsc{unless}\ \neg q}{p \ \textsc{needs}\ r} \qquad \text{(weak-transitivity)} \qquad \mathbf{N6}$$

Properties of AFTER

$$p \ \textsc{after}\ \neg p \qquad \text{(anti-reflexivity)} \qquad \mathbf{A1}$$

$$(p \ \textsc{after}\ q) \ \wedge \ (p \ \textsc{after}\ r) \ \equiv \ p \ \textsc{after}\ (q \ \wedge \ r) \qquad (\wedge \text{ equivalence}) \qquad \mathbf{A2}$$

$$\frac{p \ \textsc{after}\ q \qquad \Box\,(q \rightarrow r)}{p \ \textsc{after}\ r} \qquad (\textsc{after} \text{ - } \rightarrow) \qquad \mathbf{A3}$$

$$\frac{p \ \textsc{after}\ q \qquad \textsc{init}(\neg p)}{\Box(\neg q) \ \rightarrow \Box(\neg p)} \qquad (\textsc{after} \text{ - obligation}) \qquad \mathbf{A4}$$

$$p \ \textsc{after}\ q \ \equiv \ p \ \textsc{needs}\ \hat{\bigcirc} q \qquad \mathbf{A5}$$

Properties of CAUSES

$$p \ \hookrightarrow \ p \qquad \text{(reflexivity)} \qquad \mathbf{C1}$$

$$\frac{p \ \hookrightarrow \ q}{\Box(\neg q) \ \rightarrow \Box(\neg p)} \qquad (\hookrightarrow \text{ - obligation}) \qquad \mathbf{C2}$$

$$\frac{p \ \hookrightarrow \ q \qquad q \ \rightarrow \ r}{p \ \hookrightarrow \ r} \qquad (\hookrightarrow \text{ - } \rightarrow) \qquad \mathbf{C3}$$

$$p \ \mapsto \ q \ \equiv \ [(p \wedge q) \ \textsc{unless}\ (\neg p \vee \Diamond q)] \ \wedge \ p \ \hookrightarrow \ q \qquad \mathbf{C4}$$

$$\frac{p \ \mapsto \ q}{p \ \hookrightarrow \ q} \qquad \mathbf{C5}$$

$$\frac{p \ \hookrightarrow \ q \qquad q \ \mapsto \ r}{p \ \hookrightarrow \ r} \qquad \mathbf{C6}$$

Table 5: Properties of the new operators. In **N2** and **A2** we require q and r to share only variables that appear also in p.

results in too restrictive a condition, and UNLESS forces the introduction of a negation, making the formula less explicit.

$$(\neg\, \mathit{reserved}(P,F)) \text{ UNLESS } (\, \mathit{seats\_available}(F) \;+\; \mathit{request}(P,F)\,)$$

The properties of AFTER listed in table 5 are similar to those of NEEDS: it is anti-reflexive (A1), distributes over the conjunction (A2), it shows a transitive closure (A3), and entails an obligation (A4). Finally, A5 relates AFTER and NEEDS.

3.3.3 Causes

To describe CAUSES, we compare it with the stronger $\mapsto$. First of all, the following equivalence holds:

$$p \mapsto q \quad \equiv \quad p \hookrightarrow q \;\wedge\; [(p \wedge q) \text{ UNLESS } (\neg\, p \vee \Diamond q)]$$

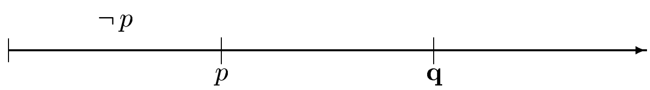
In other words, to satisfy a property expressed with $\mapsto$ we cannot falsify the conclusions without also falsifying the premises or ensuring that the conclusion will be verified once more. This is why the use of $\hookrightarrow$ is very useful in the earliest stages of a refinement, when it is more convenient to say something like: "if p appears, then ..." deferring the choices related to p staying true or not. For instance, let us consider:

$$\mathit{request}(R) \quad \hookrightarrow \quad \mathit{answer}(R,A)$$

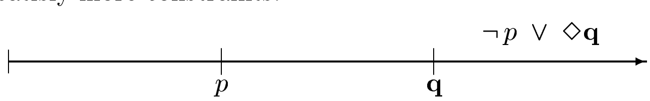
Its premise says that there is a request R. The consequence states that there is an answer A to R. The formula itself states that, following the state in which the premise *becomes* true, the process will reach a state, in which the consequence holds: nothing else is implied on future evolutions.

Note that, in tuple spaces, the natural way to falsify a condition is to consume a tuple: for instance, once the answer as been taken into consideration, it is natural to consume it. With the formula above, the refiner is free to follow this line. If, on the contrary, we use $\mapsto$, we force the refiner to consume also some piece of the premises, when he wants to consume the consequence.

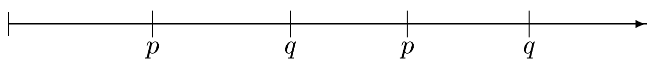
To sum up, we show models of $p \mapsto q$ and of $p \hookrightarrow q$. The property $p \hookrightarrow q$ requires a state satisfying the conclusion (q) to follow the states in which the premise (p) *becomes* true. For instance, the following computation[8] is a model for $p \hookrightarrow q$:



i.e. each state in which p becomes true is followed by a state in which q is true. There are no other constraints: for instance, p can stay true even after the state in which q is true, without requiring q to stay or become true again. Instead, a computation which is a model for $p \mapsto q$ has to satisfy more constraints:



A state in which p is true is followed by one in which q is true: to let q undetermined after it appeared, we have to negate p. If p continues to be true after q became true, then further states with q true are required:



[8]We are using the same conventions introduced in the caption of Table 4.

The properties of $\hookrightarrow$ are also listed in table 5: it is reflexive (C1), entails an obligation (C2), and shows a transitive closure (C3). Some properties relate $\hookrightarrow$ and $\mapsto$: the first operator is weaker than the second one (C4 and C5); they can be combined to derive a transitivity property (C6).

4 Semantics

We define both an axiomatic and an operational semantics for the prototype language defined in section 2. The axiomatic semantics is in the weakest precondition style, and the transition system defining the operational semantics is based on the new notion of enabling precondition, which exploits the angelic choice in non-deterministic selection.

We relate operational and weakest precondition semantics so that the properties derived using the axiomatic semantics are correct and complete with respect to the operational behaviour. This result overcomes the problems related to the basic statement, which is a non-deterministic rule rather than a simple assignment.

4.1 Motivations

In deterministic sequential languages the operational semantics of a program, given an initial state, is the singleton set containing the only possible computation. It is not so difficult to prove that a program satisfies the specification, i.e. to prove assertions of the kind $\{p\}$ *program* $\{q\}$. The calculus is usually based on a weakest precondition semantics. Its correctness is ensured by the following result. Let $T(S)$ be the transition system defining the operational semantics of an atomic statement S:

$$\langle s, s' \rangle \in T(S) \text{ iff } \forall q\, [s' \| \!\!= q \leftrightarrow s \| \!\!= wp(S, q)] \tag{1}$$

Then, the sequential combination $S_1; S_2$ corresponds to $wp(S_1, wp(S_2, q))$ and so on [8].

In most concurrent languages (e.g. Action Systems [10], Unity [21]) the operational semantics is a (possibly infinite) set of computations. All the computations start from the same initial state s_0. Hence, we can think of the set of computations as the set of branches of a tree T rooted in s_0. Let s be a node of the tree, and $s_1, \ldots, s_n$ the sons of s. A transition from s to s_i is defined by one of the (deterministic) statements (S_i) that can be executed in s. In other words, there is non-determinism in statement selection, while after the commitment to a particular statement, the behaviour is deterministic. For deterministic statements result (1) relating operational and axiomatic semantics still holds. Verification calculi for these languages are based on it. Their novelty is the way of combining assertions of the kind $\{p\}$ *statement* $\{q\}$ to prove that programs satisfy specifications even in case of non-deterministic choice, and we do the same (see, for instance [21]).

In tuple space languages (1) does not hold. Statements are rules and rule behaviour is non-deterministic. A rule, when executed in a state s can lead to different states, depending upon which atoms in s are selected to be read and consumed by rule guard.

Example 4.1 *Consider the rule:*

$$R: \ \textbf{cons}\ d(\mathrm{X}) \ \ \textbf{ask}\ \mathrm{X} > 0 \ \ | \ \ \textbf{body}\ \ \mathrm{Y}\, is\, \mathrm{X}-1. \ \ \textbf{tell}\ d(\mathrm{Y})$$

The intended meaning is that, given $s = \{d(2), d(4)\}$, transitions:

$$s \to \{d(1), d(4)\} \qquad \textit{and} \qquad s \to \{d(2), d(3)\}$$

are allowed, and therefore $\{d(1), d(4)\} \in T(R)(s)$. On the other side, given $q = d(1) + d(4)$, $s \| \!\!\not= wp(R, q)$, since from s we may end in $\{d(2), d(3)\}$ that does not satisfy q.

Non-determinism has been studied by Back and von Wright in [11]. We summarize below the results of interest. In section 4.3 we use them to define the weakest precondition of a rule, and, in section 4.4, to introduce the notion of enabling precondition, and to define an operational semantics.

According to Back and von Wright, a *demonic strict* assignment has the following syntax:

$$\wedge\, x := x'.c$$

where x, x' are variables, and c is a predicate on x'. The statement $x := x'.c$ assigns to x any value x' satisfying c. As an example, $x := x'.\{x' \in \mathbb{Z}\}$ assigns to x any integer. The weakest precondition of a strict demonic assignment statement is defined as follows.

$$wp(\wedge\, x := x'.c,\, q) = \exists x'.c \;\;\wedge\;\; \forall x'.(c \rightarrow q[x'/x])$$

The weakest precondition ensures that a state satisfying the postcondition is reached even with a *demonic* choice of the value x' satisfying c, i.e. one that tries to abort the statement. For instance:

$$wp(\wedge\, x := x'.\{x' \in \mathbb{N}\},\, x \geq 0) = true \qquad wp(\wedge\, x := x'.\{x' \in \mathbb{N}\},\, x \neq 7) = false$$

In the case of *angelic strict* assignment we have:

$$wp(\vee\, x := x'.c, q) = \exists x'.(c \;\wedge\; q[x'/x])$$

If the weakest precondition is satisfied, then a state satisfying the postcondition is reached with an angelic choice of the value x' satisfying c, i.e. one that tries to execute the statement. For instance:

$$wp(\vee\, x := x'.\{x' \in \mathbb{N}\},\, x \geq 0) = true \qquad wp(\vee\, x := x'.\{x' \in \mathbb{N}\},\, x = 4) = true$$

The only sensible choice for the weakest precondition of a rule is the demonic behaviour since it ensures that a state satisfying the postcondition is reached from the state satisfying the precondition, independently from the choices in the rule. The angelic behaviour will be introduced to define the operational semantics in section 4.4.

Before going into the technicalities, we need to compare our work with Swarm [26]. Unity influences the proof logic for Swarm as well. Like in Unity, the building blocks of the proofs in Swarm are Hoare triples. However, Swarm is a tuple space language, and the basic statement is read-write statement on the tuple space rather than an assignment. Consequently, the usual proof rule to derive assertions of the kind $\{p\}\, R\, \{q\}$ cannot be used. In Swarm, a transaction relation predicate *step* is defined: $step(t, R, t')$ means that rule R is in tuple space t, and its execution can transform t in t'. The meaning of the assertion $\{p\}\ R\ \{q\}$ is then supplied saying:

$$\{p\}\ R\ \{q\} \;\equiv\; \forall t, t'.\ step(t, R, t') :\ p(t) \Rightarrow q(t')$$

However, a constructive derivation for Hoare triples in the case of rules was missing. The definition of a formal method to prove $\{p\}\, R\, \{q\}$, allowing this *gap* to be closed, is one of the results of our work, the difficulty being caused by the non-deterministic features of the rule discussed above. Moreover, Hoare triples are too strong when proving liveness properties. When refining liveness, we do not need to require that:

$$\forall t, t'.\ step(t, R, t') :\ p(t) \Rightarrow q(t')$$

In fact, it is sufficient that a fair selection is guaranteed and to show that:

$$\forall t.\, p(t) :\ \exists t'.\ step(t, R, t')\ \wedge\ q(t')$$

We introduce the notion of rule *enabling precondition*, capturing this condition, and provide a formal derivation for it. When we say that $p \rightarrow ep(R,\, q)$, we say that rule R can fire in every state s satisfying p and *possibly* lead to state s' satisfying q. Note that (1) says that wp and ep coincide in the case of deterministic statements, while in the non-deterministic ones ep is implied by wp (see proposition 4.19 (pag. 27)).

4.2 Preliminaries

We need some preliminaries. The reader can skip this section now, and use it as a reference while looking at the next ones. We use here a different notation for the state formulae introduced in section 3.1. The notation introduced here is less readable, but suits better to the topic of this section.

A property of a state is here a formula built over:

1. Conditions $p(\bar{\mathrm{X}})^{\bullet\, n}$ with n natural number and $\bullet \in \{>, \geq, =, \leq, <\}$, constraining the multiplicity of atom $p(\bar{\mathrm{X}})$ in the state. Here and in the following $\bar{\mathrm{X}}$ indicates an array of variables.
2. Conditions $\mathrm{X} \neq \mathrm{Y}$, $\mathrm{X} = \mathrm{Y}$, $\mathrm{X}\, is\, \mathrm{Y} + 3\ \ldots$ constraining variables X, Y appearing in the other condition (primitives inherited from Prolog.)
3. Classical connectives and quantifiers of first order logic

The semantics is based on definition 3.8 (pag. 13). In particular, we require all the variables occurring in a formula p to be bound in order to say $s \| \!\models p$. The operators defined here, as well as the wp derivation, are syntax driven, and can be easily automated.

The following transformations are based on the fact that the multiplicity of an atom $a(\bar{\mathrm{X}})$ appearing in the postcondition is affected by a tell (consume) operation **tell** $t(\bar{\mathrm{B}})$ (**cons** $t(\bar{\mathrm{B}})$) only when $a = t$ and the arguments $\bar{\mathrm{X}}$ and $\bar{\mathrm{B}}$ coincide.

Definition 4.2 (Atomic unifying transformation) *Let $a(\bar{\mathrm{X}})$, $t(\bar{\mathrm{B}})$ be atoms with no variables in common, we define their atomic unifying transformation as follows.*

$$aut(a(\bar{\mathrm{X}})^{\bullet\, n},\, t(\bar{\mathrm{B}})) = \begin{cases} a(\bar{\mathrm{X}})^{\bullet\, n} & if\; a \neq t \\ [t(\bar{\mathrm{B}})^{\bullet\, n}\ \wedge\ \bar{\mathrm{X}} = \bar{\mathrm{B}}] \vee [t(\bar{\mathrm{X}})^{\bullet\, n}\ \wedge\ \bar{\mathrm{X}} \neq \bar{\mathrm{B}}] & if\; a = t \end{cases}$$

Definition 4.3 (Unifying transformation) *Let q be a state property, $t(\bar{\mathrm{B}})$ an atom:*

$$ut(q,\, t(\bar{\mathrm{B}}))\ =\ q[a(\bar{\mathrm{X}})^{\bullet\, n} / aut(a(\bar{\mathrm{X}})^{\bullet\, n},\, t(\bar{\mathrm{B}}))]\ \ \textit{for every atom } a(\bar{\mathrm{X}}) \textit{ in } q$$

Definition 4.4 (Unfolding) *The unfolding of a list L of atoms is defined as the disjunction of all the possible unifications of the atoms in L, i.e. as the disjunction of all the partitions of L in multisets of identical atoms. For instance:*

$$\begin{array}{ll} unf(t(\bar{\mathrm{X}})) = & t(\bar{\mathrm{X}})^{\geq 1} \\ unf(t(\bar{\mathrm{X}}),\, t(\bar{\mathrm{Y}})) = & [t(\bar{\mathrm{X}})^{\geq 2} \wedge \bar{\mathrm{X}} = \bar{\mathrm{Y}}]\ \vee\ [t(\bar{\mathrm{X}})^{\geq 1} \wedge t(\bar{\mathrm{Y}})^{\geq 1} \wedge \bar{\mathrm{X}} \neq \bar{\mathrm{Y}}] \end{array}$$

We translate rule guards in state formulae. Without loss of generality, we consider rule guards with the structure CONSUME ASK (i.e. all the consume conditions preceding the ask conditions), and with the consume conditions sorted by name (of the predicate). Then the next definition applies:

Definition 4.5 (Guards to state formulae)

$$g2f(G) = \forall\bar{\mathrm{N}}.\,[a2f(\textsc{ask}) \;\wedge\; c2f(\textsc{consume})] \quad \text{with:}$$

$$\begin{array}{ll}
a2f(\textsc{ask},\, \mathbf{ask}\; t(\bar{\mathrm{B}})) & = a2f(\textsc{ask}) \;\wedge\; t(\bar{\mathrm{B}})^{\geq 1} \\
a2f(\textsc{ask},\, \mathbf{ask}\; \neg\, t(\bar{\mathrm{B}})) & = a2f(\textsc{ask}) \;\wedge\; t(\bar{\mathrm{B}})^{=0} \\
a2f(\textsc{ask},\, \mathbf{ask}\; \mathrm{X}{=}\mathrm{Y}\; (\mathrm{X}{\neq}\mathrm{Y}\ldots)) & = a2f(\textsc{ask}) \;\wedge\; \mathrm{X}{=}\mathrm{Y}\; (\mathrm{X}{\neq}\mathrm{Y}\ldots)
\end{array}$$

$$c2f(\textsc{consume},\, \mathbf{cons}\; t(\bar{\mathrm{B}}_1)\; \ldots\; \mathbf{cons}\; t(\bar{\mathrm{B}}_m)) = c2f(\textsc{consume}) \;\wedge\; unf(t(\bar{\mathrm{B}}_1),\, \ldots,\, t(\bar{\mathrm{B}}_m))$$

Example 4.6 (Guards to state formulae)

$$g2f(\mathbf{ask}\; p(\bar{\mathrm{X}})\; \mathbf{ask}\; p(\bar{\mathrm{Y}})\;\; \mathbf{ask}\; \bar{\mathrm{X}} > 0) \;=\; p(\bar{\mathrm{X}})^{\geq 1} \wedge p(\bar{\mathrm{Y}})^{\geq 1} \wedge \bar{\mathrm{X}} > 0$$
$$g2f(\mathbf{cons}\; p(\bar{\mathrm{X}})\; \mathbf{cons}\; p(\bar{\mathrm{Y}})\; \mathbf{ask}\; \bar{\mathrm{X}} \geq \bar{\mathrm{Y}}) \;=\; [p(\bar{\mathrm{X}})^{\geq 2} \wedge \bar{\mathrm{X}} = \bar{\mathrm{Y}}] \;\vee\; [p(\bar{\mathrm{X}})^{\geq 1} \wedge p(\bar{\mathrm{Y}})^{\geq 1} \wedge \bar{\mathrm{X}} > \bar{\mathrm{Y}}]$$
$$g2f(\mathbf{ask}\; p(\bar{\mathrm{X}})\; \mathbf{cons}\; p(\bar{\mathrm{Y}})\; \mathbf{cons}\; p(\bar{\mathrm{Y}})\;\; \mathbf{ask}\; \neg\, q(\bar{\mathrm{Y}})) \;=\; p(\bar{\mathrm{Y}})^{\geq 2} \wedge q(\bar{\mathrm{Y}})^{=0} \;\wedge\; p(\bar{\mathrm{X}})^{\geq 1}$$

4.3 Weakest Precondition Semantics

We define in this section the weakest precondition semantics for the prototype language of section 2. In the next section we complete the axiomatic semantics by introducing the enabling precondition semantics, exploit it to define an operational semantics, and relate operational and axiomatic semantics.

We define the weakest precondition of a rule. Let R be the rule: GUARD | BODY TELL_1; TELL_2 with ASK and CONSUME lists of ask and consume conditions of the guard, respectively, and q be a state property. Then the next definition applies (side numbers refer to the comments below). In table 6 we derive the weakest precondition of the rule of example 4.1 (pag. 18) to end up in a state satisfying $d(\mathrm{Z})$.

Definition 4.7 (Rule weakest precondition)

$$\begin{array}{l}
wp(R, q) = \\
\quad wp(\textsc{guard},\, wp(\textsc{consume},\, wp(\textsc{body},\, \{wp(\textsc{tell}_1,\, q),\, wp(\textsc{tell}_2,\, q)\})))
\end{array} \qquad (1)$$

$$wp(\mathbf{tell}\; t(\bar{\mathrm{B}}),\; q) = ut(q, t(\bar{\mathrm{B}}))\, [t(\bar{\mathrm{B}})^{\bullet\, n}/t(\bar{\mathrm{B}})^{\bullet\, n-1}] \qquad (2)$$

$$wp(\mathbf{body}\; t(\bar{\mathrm{B}}),\; \{q_1,\, q_2\}) = [t(\bar{\mathrm{B}}) \;\wedge\; q_1] \;\;\vee\;\; [\neg t(\bar{\mathrm{B}}) \;\;\wedge\;\; q_2] \qquad (3)$$

$$wp(\mathbf{cons}\; t(\bar{\mathrm{B}}),\; q) = ut(q, t(\bar{\mathrm{B}}))[t(\bar{\mathrm{B}})^{\bullet\, n}/t(\bar{\mathrm{B}})^{\bullet\, n+1}] \qquad (4)$$

$$wp(\textsc{guard},\; q) = g2f(\textsc{guard}) \;\wedge\; [\, g2f(\textsc{guard}) \rightarrow q\,] \qquad (5)$$

(1) *Rule weakest precondition:* rule R and property q do not have to share any variable. (It is always possible to rename rule variables to satisfy this constraint.) We recall that postconditions TELL_1 and TELL_2 apply in case of successful or unsuccessful evaluation of the body, respectively.

(2) *Tell weakest precondition:* **tell** $t(\bar{\mathrm{B}})$ is an atomic tell condition. Unifying transformation $ut(q, t(\bar{\mathrm{B}}))$ is defined in 4.2. The wp of the whole TELL is the wp of the sequential concatenation of its atomic parts. The result is independent from the order. ($\bar{\mathrm{B}}, \bar{\mathrm{A}}, \ldots$ indicate arrays of variables.) As an example:

$$wp(\mathbf{tell}\; d(\mathrm{Y}),\; d(\mathrm{Z})^{\geq 2}) \;=\; [d(\mathrm{Z})^{\geq 1} \wedge \mathrm{Y}{=}\mathrm{Z}] \vee d(\mathrm{Z})^{\geq 2}$$

In fact, either $\mathrm{Y}{=}\mathrm{Z}$ or $\mathrm{Y}{\neq}\mathrm{Z}$: in the first case only one occurrence of $d(\mathrm{Z})$ is needed before the tell operation (the other one is written); in the second case two occurrences of $d(\mathrm{Z})$ are needed before the tell.

(3) *Body weakest precondition:* **body** $t(\bar{\mathrm{B}})$ is the body. The two postconditions refer to successful evaluation of the body, or failure, respectively.

$$wp(\textbf{body}\ p(\mathrm{Y},\mathrm{Z}),\ \{d(\mathrm{X},\mathrm{Z})^{\geq 2},\ a(\mathrm{X})^{\geq 1}\}) \ = \ [p(\mathrm{Y},\mathrm{Z}) \wedge d(\mathrm{X},\mathrm{Z})^{\geq 2}] \ \vee \ [\neg p(\mathrm{Y},\mathrm{Z}) \wedge a(\mathrm{X})^{\geq 1}]$$

If Z is the first computed answer substitution for $p(\mathrm{Y},\mathrm{Z})$, then $d(\mathrm{X},\mathrm{Z})^{\geq 2}$ has to hold before body evaluation, otherwise, if no computed answer substitution for the body exists, i.e., the body fails, then $a(\mathrm{X})^{\geq 1}$ is the needed precondition. Note that we are considering that the BODY, being defined by a Prolog program, is deterministic: BODY evaluation returns the first computed substitution, if any.

(4) *Consume weakest precondition:* **cons** $t(\bar{\mathrm{B}})$ is an atomic consume condition. As for tell: $ut(q, t(\bar{\mathrm{B}}))$ is defined in 4.2; the wp of the whole CONSUME is the wp of the sequential concatenation of its atomic parts. The result is independent from the order in this case too. As an example:

$$wp(\textbf{cons}\ d(\mathrm{Y}),\ d(\mathrm{Z})^{\geq 2}) \ = \ [d(\mathrm{Z})^{\geq 2} \wedge \mathrm{Y} \neq \mathrm{Z}] \ \vee \ d(\mathrm{Z})^{\geq 3}$$

in fact if $\mathrm{Y} \neq \mathrm{Z}$ then only two occurrences of $d(\mathrm{Z})$ are needed before the tell operation, while if $\mathrm{Y} = \mathrm{Z}$, i.e. an occurrence of $d(\mathrm{Z})$ is consumed, then three occurrences of $d(\mathrm{Z})$ are needed before the consume.

(5) *Guard weakest precondition:* $g2f(\mathrm{GUARD})$ is the translation, in terms of state formulae, of the constraints expressed by the guard, according to definition 4.5; $\bar{\mathrm{X}}$ are the variables occurring free in the guard translation. Let **cons** $d(\mathrm{Y})$, **ask** $\mathrm{Y} > 0$ be a rule guard G.

$$wp(G,\ b(\mathrm{Y},\mathrm{Z})^{\geq 2}) \ = \ [d(\mathrm{Y})^{\geq 1} \wedge \mathrm{Y} > 0] \ \wedge \ [(d(\mathrm{Y})^{\geq 1} \wedge \mathrm{Y} > 0) \ \rightarrow b(\mathrm{Y},\mathrm{Z})^{\geq 2}]$$

in other words, there must be an atom unifying with the rule guard and for every unifying instance of Y, two occurrences of $b(\mathrm{Y},\mathrm{Z})$ are needed as precondition.

The weakest precondition of a guard is based on the weakest precondition of the demonic strict assignment. In fact, if we apply function t of definition 3.6, we have:

$$t(wp(\mathrm{GUARD},\ q)) = \exists \bar{\mathrm{X}}.g2f(\mathrm{GUARD}) \ \wedge \ \forall \bar{\mathrm{X}}.[\,g2f(\mathrm{GUARD}) \rightarrow q\,]$$

Special cases. Let us now consider two special cases of rule.

a) A rule can have a non empty body and only one tell, i.e. have the form GUARD | BODY TELL (this is actually the case of the rule in table 6). The meaning is that the body cannot fail. Let the body be $t(\bar{\mathrm{B}})$, then $wp(R, q)$ reduces to:

$$wp(R,q) = wp(\mathrm{GUARD},\ wp(\mathrm{CONSUME},\ wp(\mathrm{BODY},\ \{wp(\mathrm{TELL},\ q), false\})))$$

and

$$wp(\textbf{body}\ t(\bar{\mathrm{B}}),\ \{q', false\}) \ = \ (t(\bar{\mathrm{B}}) \wedge q') \ \vee \ (\neg t(\bar{\mathrm{B}}) \wedge false) \ = \ t(\bar{\mathrm{B}}) \wedge q'$$

b) In the second case R has the form GUARD | TELL , i.e. it has an empty body, or, equivalently the body is *true*. and thus omitted. In this case the weakest precondition of the body reduces to the identity function, and:

$$wp(R,q) \ = \ wp(\mathrm{GUARD},\ wp(\mathrm{CONSUME},\ wp(\mathrm{TELL},\ q)))$$

$$\mathrm{wp}(R, d(\mathrm{Z})) \;=\; d(\mathrm{X}) + \mathrm{X} > 0 \;\;\wedge \;\; [d(\mathrm{X}) + \mathrm{X} > 0] \;\rightarrow\; \{[\mathrm{X} = \mathrm{Z} + 1] \;\vee\; [d(\mathrm{Z}) + d(\mathrm{Z})] \;\vee\; [d(\mathrm{Z}) + \mathrm{Z} \leq 0]\}$$

Derivation:

$d(\mathrm{Z})^{\geq 1}$

tell $d(\mathrm{Y})$

$(d(\mathrm{Y})^{\geq 0} \wedge \mathrm{Z} = \mathrm{Y}) \;\vee\; (d(\mathrm{Z})^{\geq 1} \wedge \mathrm{Y} \neq \mathrm{Z}) \quad\equiv\quad \mathrm{Z} = \mathrm{Y} \;\vee\; d(\mathrm{Z})^{\geq 1}$

body $\mathrm{Y}\, is\, \mathrm{X}-1$

$\mathrm{Y}\, is\, \mathrm{X}-1 \;\vee\; d(\mathrm{Z})^{\geq 1} \quad\equiv\quad \mathrm{Y} = \mathrm{X}-1 \;\vee\; d(\mathrm{Z})^{\geq 1}$

cons $d(\mathrm{X})$

$\mathrm{Z} = \mathrm{X}-1 \;\vee\; (d(\mathrm{Z})^{\geq 1} \wedge \mathrm{X} \neq \mathrm{Z}) \;\vee\; (d(\mathrm{X})^{\geq 2} \wedge \mathrm{X} = \mathrm{Z})$

$g2f(\textbf{cons}\; d(\mathrm{X}) \;\; \textbf{ask}\; \mathrm{X} > 0) \quad=\quad d(\mathrm{X})^{\geq 1} \wedge \mathrm{X} > 0$

$$\begin{aligned} wp(R, d(\mathrm{Z})^{\geq 1}) = \;& [d(\mathrm{X})^{\geq 1} \wedge \mathrm{X} > 0] \;\wedge \\ & (d(\mathrm{X})^{\geq 1} \wedge \mathrm{X} > 0) \rightarrow \{[\mathrm{X}-1 = \mathrm{Z}] \;\vee\; [d(\mathrm{X})^{\geq 2} \wedge \mathrm{X} = \mathrm{Z}] \;\vee\; [d(\mathrm{Z})^{\geq 1} \wedge \mathrm{X} \neq \mathrm{Z}]\} \\ = \;& [d(\mathrm{X})^{\geq 1} \wedge \mathrm{X} > 0] \wedge \\ & (d(\mathrm{X})^{\geq 1} \wedge \mathrm{X} > 0) \rightarrow \{[\mathrm{X} = \mathrm{Z} + 1] \;\vee\; d(\mathrm{Z})^{\geq 2} \;\vee\; [d(\mathrm{Z})^{\geq 1} \wedge \mathrm{Z} \leq 0]\} \end{aligned}$$

In words, a state s, enabling rule R and guaranteeing that the execution of R leads to a state containing $d(\mathrm{Z})$, must both contain an atom satisfying the guard and satisfy at least one of the following conditions:

a. there are only atoms $d(\mathrm{Z}+1)$ having predicate d and argument > 0 (one of them is surely taken leading to $d(\mathrm{Z})$);

b. there are two atoms $d(\mathrm{Z})$ (even if one is taken, one is left);

c. there are some (one at least) occurrences of $d(\mathrm{Z})$, but $\mathrm{Z} \leq 0$, so they stay there.

Let us finally observe that, according to the notation introduced at the beginning of section 4.2, the weakest precondition of $d(\mathrm{Z})$ is exactly the one given on the top of the page.

Table 6: Derivation of a weakest precondition

Example 4.8 *Another interesting example is the weakest precondition of a rule R to end up in a state satisfying true: it reduces to require that the rule guard is satisfied. In fact:*

$$\begin{aligned} wp(R, true) &= wp(\text{GUARD}, wp(\text{CONSUME}, wp(\text{BODY}, \{wp(\text{TELL}_1, true), wp(\text{TELL}_2, true)\}))) \\ &= wp(\text{GUARD}, wp(\text{CONSUME}, wp(\text{BODY}, \{true, true\}))) \\ &= wp(\text{GUARD}, wp(\text{CONSUME}, true)) \\ &= wp(\text{GUARD}, true) \\ &= g2f(\text{GUARD}) \end{aligned}$$

On the other side, the weakest precondition of a rule to end up in a state satisfying $false$ is $false$.

4.3.1 Properties

The following results state some properties of the weakest precondition.

Proposition 4.9 *Let p and q be state formulae and R a rule. Then*

$$wp(R, p \wedge q) \equiv wp(R, p) \wedge wp(R, q)$$

Proof. in the appendix.

Corollary 4.10 *Let p and q be state formulae and R a rule. Then:*

$$\frac{p \rightarrow q}{wp(R, p) \rightarrow wp(R, q)}$$

Corollary 4.11 *Let p be a state formula and R a rule, then:*

$$wp(R, \neg p) \rightarrow \neg wp(R, p)$$

Corollary 4.12 *Let p and q be state formulae and R a rule, then :*

$$wp(R, p) \vee wp(R, q) \rightarrow wp(R, p \vee q)$$

4.4 Towards an Operational Semantics: Enabling Preconditions

We need to define an operational semantics for our language, to describe the allowed computations of a system. We define it in terms of transitions. As discussed in section 4.1, we cannot define the transitions of a rule R as the pairs of states

$$\langle s, s' \rangle \text{ s.t. } \forall q\, [s' \Vdash q \leftrightarrow s \Vdash wp(S, q)]$$

because of the non-determinism of the rules.

Hence, we introduce the new notion of enabling precondition (ep), show how to derive the enabling precondition of a rule, and base the operational semantics on it. The enabling precondition actually captures the operational behaviour of a rule and overcomes the problem discussed in section 4.1. For instance, consider rule R of example 4.1 (pag. 18), and take $q = d(1)^{\geq 1} \wedge d(4)^{\geq 1}$. We have:

$$\{d(2), d(4)\} \not\Vdash \ wp(R, q)$$

while we will find that

$$\{d(2), d(4)\} \Vdash \ ep(R,\, q)$$

In the section 4.6 we relate weakest and enabling preconditions.

4.4.1 Enabling Precondition Semantics

Let q be a state property, R a rule: GUARD | BODY TELL$_1$; TELL$_2$ with ASK and CONSUME lists of ask and consume conditions of the guard, respectively. Then the following definition applies. We make use here of some notions introduced in sections 4.2 and 4.3, in particular we refer to definition 4.7 (pag. 21) and to definition 4.5 (pag. 21).

Definition 4.13 (Rule enabling precondition)

$$ep(R, q) = ep(\text{GUARD},\, wp(\text{CONSUME},\, wp(\text{BODY},\, \{wp(\text{TELL}_1,\, q),\, wp(\text{TELL}_2,\, q)\})))$$

$$ep(\text{GUARD},\ q) = \ g2f(\text{GUARD}) \ \wedge \ q$$

with the constraints (1), and (5) of definition 4.7 (pag. 21). We observe that the enabling precondition of a guard is based on the weakest precondition of the angelic strict assignment. In fact, if we apply function t of definition 3.6, we have:

$$t(ep(\text{GUARD},\ q)) \ = \ t(g2f(\text{GUARD}) \ \wedge \ q) \ = \ \exists\bar{\text{X}}.[\, g2f(\text{GUARD}) \ \wedge \ q\,]$$

Example 4.14 *Let* **cons** $d(\text{Y})$, **ask** $\text{Y} > 0$ *be a rule guard* G*.*

$$ep(G,\, b(\text{Y},\text{Z})^{\geq 2}) \ = \ d(\text{Y})^{\geq 1} \wedge \text{Y} > 0 \ \wedge \ b(\text{Y},\text{Z})^{\geq 2}$$

in other words, there must be an atom unifying with the rule guard and two occurrences of $b(\text{Y},\text{Z})$ *for the corresponding unifying instance of* Y*. A further example is in table 7, and refers to rule* R_2 *of example in table 2.*

The definition is simplified if rule R has form GUARD | BODY TELL or GUARD | TELL. In the first case ep reduces to $ep(\text{GUARD},\, wp(\text{CONSUME},\, wp(\text{BODY},\, wp(\text{TELL},\, q))))$, while in the second case it reduces to $ep(\text{GUARD},\, wp(\text{CONSUME},\, wp(\text{TELL},\, q)))$

Let us consider, for instance, rule R of example 4.1 (pag. 18): $ep(R, d(\text{Z})^{\geq 1}) \ =$

$$\exists\text{X}.\, [d(\text{X})^{\geq 1} \wedge \text{X} > 0 \wedge (\text{X} = \text{Z} + 1 \ \vee \ (d(\text{Z})^{\geq 1} \wedge \text{X} \neq \text{Z}) \ \vee \ d(\text{Z})^{\geq 2})]$$

The enabling precondition requires that the rule guard is satisfied, i.e. $\exists\text{X}.\, [d(\text{X})^{\geq 1} \wedge \text{X} > 0]$. Moreover, the tuple space has to contain:

a. an atom $d(\text{Z}+1)$, with $\text{Z}+1 > 0$: $\exists\text{X}.\, [d(\text{X})^{\geq 1} \wedge \text{X} > 0 \wedge \text{X} = \text{Z}+1]$; or

b. two atoms $d(\text{Z})$: $\exists\text{X}.\, [d(\text{X})^{\geq 1} \wedge \text{X} > 0 \wedge d(\text{Z})^{\geq 2}]$; or

c. one atom $d(\text{Z})$, with $\text{Z} \neq \text{X}$: $\exists\text{X}.\, [d(\text{X})^{\geq 1} \wedge \text{X} > 0 \wedge d(\text{Z})^{\geq 1} \wedge \text{X} \neq \text{Z}]$.

If none of these condition is satisfied, it is not possible to end up in a state containing $d(\text{Z})$. If we compare with the weakest precondition derived in table 6, condition b is the same in the two cases, while here conditions a and c are weaker:

let R : **cons** $n(\text{Z})$ **ask** $n(\text{Y})$ **ask** $\text{Z}\, mod\, \text{Y} = 0$ |

$$\text{ep}(R, \neg n(\text{X})) \;=\; \begin{array}{l} [n(\text{X}) + n(\text{Y}) + \neg(n(\text{X}) + n(\text{X})) \;\wedge\; \text{X}\, mod\, \text{Y} = 0] \;\vee \\ [n(\text{Z}) + n(\text{Y}) + \neg n(\text{X}) \;\wedge\; \text{Z}\, mod\, \text{Y} = 0] \end{array}$$

Derivation:

$$n(\text{X})^{=0}$$

$$\textbf{cons}\; n(\text{Z})$$

$$[\text{Z} = \text{X} \;\wedge\; n(\text{X})^{=1}] \;\vee\; [\text{X} \neq \text{Z} \;\wedge\; n(\text{X})^{=0}]$$

$$g2f(\textbf{cons}\; n(\text{Z}) \;\; \textbf{ask}\; n(\text{Y}))\;\; \textbf{ask}\; \text{Z}\, mod\, \text{Y} = 0 \quad = \quad n(\text{Z})^{\geq 1} \wedge n(\text{Y})^{\geq 1} \wedge \text{Z}\, mod\, \text{Y} = 0$$

$$\begin{aligned} ep(R, n(\text{X})^{=0}) = \;& [n(\text{Z})^{\geq 1} \;\wedge\; n(\text{Y})^{\geq 1} \;\wedge\; \text{Z}\, mod\, \text{Y} = 0 \;\wedge \\ & [(\text{Z} = \text{X} \;\wedge\; n(\text{X})^{=1} \;\vee\; (\text{X} \neq \text{Z} \;\wedge\; n(\text{X})^{=0}] \\ = \;& [n(\text{X})^{=1} \;\wedge\; n(\text{Y})^{\geq 1} \;\wedge\; \text{Z}\, mod\, \text{Y} = 0] \;\vee \\ & [n(\text{Z})^{\geq 1} \;\wedge\; n(\text{Y})^{\geq 1} \;\wedge\; n(\text{X})^{=0} \;\wedge\; \text{Z}\, mod\, \text{Y} = 0] \end{aligned}$$

Table 7: Derivation of an enabling precondition

a: in case of angelic choice, the presence of an atom $d(\text{Z}+1)$ with $\text{Z}+1 > 0$ is a sufficient condition, while if the choice is demonic all the atoms having predicate d and argument > 0 are required to equal $d(\text{Z}+1)$.

c: the presence of $d(\text{Z}) + d(\text{X})$, with $\text{X} > 0$ is not a sufficient condition if the choice is demonic: the demon would select $d(\text{Z})$. He cannot if $\text{Z} \leq 0$, as required by condition c of table 6.

4.4.2 Properties

The following statements provide some properties of the enabling precondition. Both proofs are in the appendix.

Proposition 4.15 *Let p and q be state formulae and R a rule, then :*

$$ep(R,\, p \vee q) \equiv ep(R,\, p) \vee ep(R,\, q)$$

Corollary 4.16 *Let p be a state formula and R a rule, then :*

$$\neg\, ep(R,\, p) \wedge ep(R,\, true) \;\;\rightarrow\;\; ep(R,\, \neg\, p)$$

4.5 Operational Semantics: a Transition System

We can now define the operational semantics of our language, based on enabling preconditions.

Definition 4.17 (Operational semantics)

$$T(R)(s) = \{s' \mid \forall q\, \forall \bar{\mathrm{x}}\ \ \textit{free in } q.\ \ s' \| \!\!= q(\bar{\mathrm{x}}) \rightarrow s \| \!\!= ep(R, q(\bar{\mathrm{x}}))\}$$

$T(R)$ takes a state s and returns the set of states s' such that rule R, when applied in state s, can lead to state s'. Lemma 4.18 ensures that this is a good definition: the relation $s \| \!\!= p$ is only defined if all the variables of p are bound.

Lemma 4.18 *Let R be a rule. A variable* x *is free in $ep(R, q)$ ($wp(R, q)$) if and only if it is free in q.*

Proof. Variables in $ep(R, q)$ ($wp(R, q)$) are either variables of q or variables of R. All the variables of R introduced during ep (wp) derivation are later universally or existentially quantified, since all the variables occurring in R, occur in the guard or in the body.

4.6 Axiomatic vs operational semantics

The following results relate *ep*, *wp* and operational semantics, and close the gap left open in section 4.1. In the next section, we will exploit these results to prove the correctness of our refinement calculus. All the proofs are in the appendix.

Proposition 4.19 *Let q be a state formula and R a rule, then:*

$$wp(R, q) \rightarrow ep(R, q)$$

Lemma 4.20 *Let R be a rule, then:*

$$wp(R, true) \equiv ep(R, true)$$

Proposition 4.21 *Let R be a rule and q a state formula, then:*

$$wp(R, q) \equiv \neg\, ep(R, \neg\, q) \wedge ep(R, true)$$

Proposition 4.22 *Let s be a state and R a rule, then:*

$$s \| \!\!= wp(R, true) \ \ \leftrightarrow \ \ T(R)(s) \neq \emptyset$$

Corollary 4.23 *Let s be a state and R a rule, then:*

$$s \| \!\!= ep(R, true) \ \ \leftrightarrow \ \ T(R)(s) \neq \emptyset$$

Theorem 4.24 *Let R be a rule, s a state, and q a state formula, then* $\forall \bar{\mathrm{x}}$ *free in q:*

$$s \| \!\!= ep(R, q(\bar{\mathrm{x}})) \ \ \leftrightarrow \ \ \exists s'. [s' \in T(R)(s) \ \wedge \ s' \| \!\!= q(\bar{\mathrm{x}})]$$

Theorem 4.25 *Let R be a rule, s a state, and q a state formula, then* $\forall \bar{\mathrm{x}}$ *free in q:*

$$s \| \!\!= wp(R, q(\bar{\mathrm{x}})) \ \ \leftrightarrow \ \ T(R)(s) \neq \emptyset \wedge \ \forall s'. [s' \in T(R)(s) \rightarrow s' \| \!\!= q(\bar{\mathrm{x}})]$$

5 Refinement

Our refinement calculus is based on the following facts: our specifications are interpreted on computations (definition 3.7); we can use the operational semantics to describe the allowed computations of a system; we can exploit the axiomatic semantics to define program properties; the relation between axiomatic and operational semantics stated in the previous section guarantees that using the former we can derive properties, which are correct with respect to the operational behaviour.

Notation 5.1 *In the following, we denote by* $(\mathbb{R}_1^m, P, Is)$ *the system defined by a program composed of rules* $R_1, \ldots, R_m$*, a partial order* P *defining rule priorities, and having initial state* Is*.*

We formally define computations. For the fairness assumptions we rely on the results of [29, 51]. Definition 5.6 introduces the notion of refinement.

Definition 5.2 (Computations) *A computation of a system* $(\mathbb{R}_1^m, P, Is)$ *is a sequence of states:* $s_0, s_1, \ldots, s_i, \ldots$ *such that* $Is = s_0$ *and for all* i *there exists a rule* R_j *with:*

$$s_{i+1} \in T(R_j)(s_i)$$
$$\forall R_k > R_j.\, T(R_k)(s_i) = \emptyset$$

and transition relation $T(R)$ *as defined in 4.17 (pag. 27).*

Definition 5.3 (Fair computations, wrt rule selection) *We require that rule selection is fair, i.e. that there does not exists a rule* R_j enabled *in infinite states of* c *and never* chosen, *where we define a rule* R_j *to be* enabled *in a state* s_i *if:*

$$\exists s' \in T(R_j)(s_i)$$
$$\forall R_k > R_j . T(R_k)(s_i) = \emptyset$$

and to be chosen *in* s_i *if:*

$$s_{i+1} \in T(R_j)(s_i)$$

Definition 5.4 (Fair computations, wrt tuple selection) *We require tuple selection to be fair as well: no tuple has to be* selectable *infinitely often without ever being* selected*. We define a tuple* $t(\bar{\mathrm{x}})$ *to be* selectable *in a state* s_i *if there exist a rule* R_j *enabled in* s_i*, a state* $s' \in T(R)(s_i)$*, and a formula* q *with* $s' \| \!\!= q$ *such that* $ep(R_j, q) \rightarrow t(\bar{\mathrm{x}})^{\geq 1}$*. We define* $t(\bar{\mathrm{x}})$ *to be* selected *in* s_i *if* $s_{i+1} \in T(R)(s_i)$ *for a given* R*, and there exists a formula* q *with* $s_{i+1} \| \!\!= q$ *and* $ep(R, q) \rightarrow t(\bar{\mathrm{x}})^{\geq 1}$

Definition 5.5 (Fair computations) *A computation is fair if and only if it is fair with respect to rule and tuple selection.*

Definition 5.6 System refines properties: ($(\mathbb{R}_1^m, P, Is) \triangleright S$) *A system* $(\mathbb{R}_1^m, P, Is)$ *refines a set of properties* S*, written* $(\mathbb{R}_1^m, P, Is) \triangleright S$*, when all the fair computations of* $(\mathbb{R}_1^m, P, Is)$ *are models of the formulae in* S*.*

To define the calculus we first address the refinement of a specification into a program, and then extend this result to the verification of refinement steps between heterogeneous systems. Besides, we take a compositional approach. We first define what it means for a rule to refine a property, and for an initial state to refine a property, then we show that we can use the axiomatic semantics to derive these relations. Finally, we show how to compose the derived relations to prove that a program refines a set of properties, i.e., a specification.

In the next section we deal with safety properties, and in section 5.2 with liveness. Finally, in section 5.3 we introduce heterogeneous systems and propose a methodology of refinement.

5.1 Refinement of safety properties

The common, informal definition for safety is the following: a safety property guarantees that *nothing bad* happens. A formalization of this intuition is due to Lamport [35]: let a computation c be any infinite sequence of states, and let C be a set of computations, then a property P is a safety property whenever:

$$\forall c \in C.\, [P(c) \leftrightarrow (\forall c' \text{ finite prefix of } c.\, P(c'))]$$

Very complex safety properties can be expressed, according to definition 3.5 (pag. 12). For instance,

$$\textsc{init}\, p(\bar{\mathrm{X}}) \rightarrow [q(\bar{\mathrm{Y}}, \bar{\mathrm{Z}})\, \textsc{unless}\, (r(\bar{\mathrm{X}}, \bar{\mathrm{Y}}) \wedge \neg \bigcirc b(\bar{\mathrm{Z}}))]$$

is not the worst we can write. However, in realistic cases, safety formulae are like $\textsc{init}\, p$, $p\, \textsc{unless}\, q$, $inv\, p$ ($\textsc{init}\, p \wedge \textsc{stable}\, p$), $p\, \textsc{needs}\, q$, with p and q state formulae.

We define $Is \rhd \textsc{init}\, p$ (initial state Is refines $\textsc{init}\ p$). Then, we define $R \rhd p\ \textsc{unless}\ q$ (rule R refines $p\ \textsc{unless}\ q$). In a similar way, we can define $R \rhd p\ \textsc{needs}\ q$, and $R \rhd p\ \textsc{unless}\ q$. Proposition 5.9 relates the weakest precondition calculus to the verification of safety properties, thus proving that refinement relations can be derived exploiting the axiomatic semantics.

Definition 5.7 $Is \rhd \textsc{init}\, p$ *iff* $Is |\!\!= p$.

Definition 5.8 ($R \rhd p\ \textsc{unless}\ q$) *Let* p *and* q *be state formulae, then* $R \rhd p\, \textsc{unless}\, q$ *iff* $\forall s$:

$$s \models (p \wedge \neg q) \quad \rightarrow \quad [\neg \exists s' \in T(R)(s) \ \vee \ (\forall s'.\, s' \in T(R)(s) \ \rightarrow \ s' \models (p \ \vee \ q))]$$

Proposition 5.9 *Let* p *and* q *be state formulae, then* $R \rhd p\, \textsc{unless}\, q$ *iff:*

$$(p \wedge \neg q) \ \rightarrow \ (w\,p(R,\, p \vee q) \vee \neg\, w\,p(R,\, true))$$

Proof. For all states s:

$$s \models (w\,p(R,\, p \vee q) \ \vee \ \neg\, w\,p(R,\, true)) \leftrightarrow$$

$$s \models w\,p(R,\, p \vee q) \ \vee \ s \models \neg\, w\,p(R,\, true)) \leftrightarrow$$

(theorem 4.25 and proposition 4.22, pag. 27)

$$(\forall s'.\, s' \in T(R)(s) \ \rightarrow \ s' \models (p \ \vee \ q)) \quad \vee \quad (\neg\, \exists s' \in T(R)(s)) \qquad \Box$$

In a similar way we prove:

Proposition 5.10 *Let* p *and* q *be state formulae, then:*

$$R \rhd p\, \textsc{needs}\, q \quad \text{iff} \quad e\,p(R,\, p \wedge \neg q) \ \rightarrow \ p$$

$$R \rhd p\, \textsc{after}\, q \quad \text{iff} \quad e\,p(R,\, p) \ \rightarrow \ (p \vee q)$$

As an example, consider invariants: a property p is invariant ($\textsc{inv}$) for a system if p holds in the initial state and p is kept invariant during system execution, i.e. if $\textsc{init}\ p \ \wedge \ \textsc{stable}\ p$ holds, or, equivalently, $\textsc{init}\ p \ \wedge \ p\ \textsc{unless}\ false$ holds. Then, according to proposition 5.9, $R \rhd \textsc{inv}\, p$ if $p \ \rightarrow \ (w\,p(R,\, p) \vee \neg\, w\,p(R,\, true))$, which precisely states that R will not fire, unless it leaves p untouched.

Example 5.11 *Property* $p = \forall \mathrm{Z}.\ (\ d(\mathrm{Z})^{\geq 1} \rightarrow \mathrm{Z} \geq 0\ )$ *is kept invariant by:* R : **cons** $d(\mathrm{X})$ **ask** $\mathrm{X} > 0 \mid$ **body** $\mathrm{Y}\, is\, \mathrm{X} - 1.$ **tell** $d(\mathrm{Y})$. *Indeed,* $wp(R, p) \vee \neg wp(R, true)$ *equals*

$$\{\forall \mathrm{X}.\ (d(\mathrm{X})^{\geq 1} \wedge \mathrm{X}>0) \rightarrow \forall \mathrm{Z}.\ [(\mathrm{Z} \neq \mathrm{X}-1 \wedge \mathrm{Z} \neq \mathrm{X} \wedge d(\mathrm{Z})^{\geq 1}) \rightarrow \mathrm{Z} \geq 0]\} \vee \neg \exists \mathrm{X}.(d(\mathrm{X})^{\geq 1} \wedge \mathrm{X}>0)$$

which is true whenever $\forall \mathrm{Z}.(d(\mathrm{Z})^{\geq 1} \rightarrow \mathrm{Z} \geq 0)$.

The notion of *refinement under given hypothesis* will be needed in proposition 5.14.

Definition 5.12 *Let* p, q, *and* r *be state formulae, then*

$R \rhd_r\ p\,\textsc{unless}\,q$ *iff* $\forall s$:
$$s \,\|\!\!\models (p \wedge r \wedge \neg q) \rightarrow [T(R)(s) = \emptyset \vee (\forall s'.\, s' \in T(R)(s) \rightarrow s' \,\|\!\!\models (p \vee q))]$$

$R \rhd_r\ p\,\textsc{needs}\,q$ *iff* $\forall s$:
$$s \,\|\!\!\models (r \wedge \neg p) \rightarrow (\forall s'.\, s' \in T(R)(s) \rightarrow s' \,\|\!\!\models (\neg p \vee q))$$

$R \rhd_r\ p\,\textsc{after}\,q$ *iff* $\forall s$:
$$s \,\|\!\!\models (r \wedge \neg p) \rightarrow [\forall s'.\, s' \in T(R)(s) \rightarrow s \,\|\!\!\models \neg p \vee s \,\|\!\!\models q]$$

Proposition 5.13 *Let* p, q, r *be state formulae, then:*

$R \rhd_r p\,\textsc{unless}\,q$ *iff* $(p \wedge r \wedge \neg q) \rightarrow (wp(R, p \vee q) \vee \neg wp(R, true))$

$R \rhd_r p\,\textsc{needs}\,q$ *iff* $ep(R, p \wedge \neg q) \rightarrow (p \vee \neg r)$

$R \rhd_r p\,\textsc{after}\,q$ *iff* $ep(R, p) \leftrightarrow (p \vee q \vee \neg r)$

To conclude the case of safety properties refinement, we show how to derive the general property that a system refines a set of safety properties.

Proposition 5.14 *Let* $(I\!\!R_1^m, P, Is)$ *be a system, and* S *a set of safety properties* f_i. *We have* $(I\!\!R_1^m, P, Is) \rhd S$ *when,* $\forall f_i \in S$, *and* $\forall j \in 1 \ldots m$:

$Is \rhd f_i$

$R_j \rhd_{H_j} f_i$ *with* $H_j = \{\neg\, wp(R_k, true) \mid R_k > R_j \in P\}$

Proof. The proof is standard, on the length of the computations (see definition 5.2). Priority between rules permits to add some hypothesis when proving refinement relations. In particular we know that if a rule R is applied, then none of the rules with higher priority can be applied.

Let us use again the Unity operator $\textsc{inv}$, where $\textsc{inv}\ p$ denotes the conjunction of $\textsc{init}\ p$ and $\textsc{stable}\ p$. We show, for instance, how to derive that a property p is invariant for system $(I\!\!R_1^m, \emptyset, Is)$, by proving that the initial state refines p and that $R_i \rhd_{H_i} \textsc{stable}\,p$ for every rule R_i.

Example 5.15 *Let* p *be a state formula, and let* $(I\!\!R_1^m, \emptyset, Is)$ *be a system:*

$R_1 \rhd \textsc{stable}\ p \ \ldots\ R_m \rhd \textsc{stable}\ p \rightarrow I\!\!R_1^m \rhd \textsc{stable}\ p$

$I\!\!R_1^m \rhd \textsc{stable}\ p$ and $Is \rhd \textsc{init}\ p \rightarrow (I\!\!R_1^m, \emptyset, Is) \rhd \textsc{inv}\ p$

5.2 Refinement of liveness properties

The class of liveness properties has a characterization as well. Informally, liveness guarantees that *something good eventually* happens. Formally, let C be any set of computations (generic infinite sequences of states), then a property P is a liveness property whenever [2]:

$$\forall\, c'.\text{finite prefix of a } c \in C\ \exists c'' \text{ infinite extension of } c'.\, P(c'')$$

We show here how to prove that a system refines a liveness property. Let us first observe that the distinction between the three progress operators UNTIL, $\mapsto$, and $\hookrightarrow$, is useful in the first steps of a refinement, when we refine specifications (for instance $\mapsto$ is transitive while UNTIL is not), but it is meaningless when we introduce rules: in any case we have to show that an UNTIL property is refined. Indeed in our setting, a rule only relates pairs of successive states: if a rule does *something good*, then it does it in a computation step. Then, if a step leads from a state satisfying p to a state satisfying q, it means that the strongest liveness property, p UNTIL q, is satisfied.

We hence show how to prove that a system refines an UNTIL property, and apply the same results when refining the weaker $\mapsto$ or $\hookrightarrow$.

We first define $R \rhd p\,\text{UNTIL}\,q$ (rule R refines $p\,\text{UNTIL}\,q$), and then show how to derive this relation exploiting the axiomatic semantics, with a result relating our enabling precondition calculus with program verification. The following definition only captures the *progress* part of an UNTIL property. Proposition 5.18 adds the needed safety conditions.

Definition 5.16 *($R \rhd p$ UNTIL q) Let p and q be state formulae, then $R \rhd p\,\text{UNTIL}\,q$ iff $\forall s$:*

$s \,|\!\models (p \wedge \neg q) \quad \rightarrow \quad \exists s'.[s' \in T(R)(s) \ \wedge\ s' \,|\!\models q]$

Proposition 5.17 *Let p and q be state formulae, then $R \rhd p\,\text{UNTIL}\,q$ iff*

$(p \wedge \neg q) \ \rightarrow\ e\,p(R,\, q)$

Proof. For all states s:

$s \,|\!\models e\,p(R,\, q) \leftrightarrow$ (Theorem 4.24, pag. 27)

$\exists s'.\, [s' \in T(R)(s) \ \wedge\ s' \,|\!\models q]$ □

We recall that a computation $c = s_0, \ldots, s_i, \ldots$ satisfies p UNTIL q when, for all s_i, $s_i |\!\models p$ implies $s_{i+1} |\!\models p \vee q$ and there exists j such that $s_{i+j} |\!\models q$. The following result shows how to prove that a system refines an UNTIL property.

Proposition 5.18 *Let p and q be state formulae, $(I\!R_1^m, P, Is)$ be a system, and for some i:*

1) $R_i \rhd p\,\text{UNTIL}\,q$
2) $(I\!R_1^m \setminus \{R_j < R_i\}, P, Is) \rhd p$ UNLESS q
3) $(\{R_k > R_i\}, P, Is) \rhd up(R_j, true)$ UNTIL $\neg\, up(R_j, true)$, *for all* $R_j > R_i$

Then, $(I\!R_1^m, P, Is) \rhd p$ UNTIL q

Proof. In the general case, in order for a system to refine an UNTIL property, we have to guarantee, see definition 5.6, that at least a rule R refines the *progress* (1), and that R and all the other rules satisfy the corresponding UNLESS property (2). If there is a priority order between rules, on one side we only have to require the system containing rules having equal or greater priority than R to satisfy such a safety constraint, on the other side we have to require that R will fire sometimes, i.e. that a state is reached in which $up(R_j, true) = false$, being R_j the rules having greater priority than R (3). □

In the special case of empty priority set the proposition above reduces to:

Corollary 5.19 *Let p, q be state formulae, a system $(\mathbb{R}_1^m, \emptyset, Is) \rhd p$ UNTIL q if for some i:*

1) $R_i \rhd p$ UNTIL q

2) $R_j \rhd p$ UNLESS q *for all* R_j, $j = 1 \ldots m$.

Example 5.20 *Let s be the state $\{n(1), n(2), n(3)\}$, and L the liveness property:*

$L: \; n(X) + n(Y) \wedge X = (Y+1) \quad \mapsto \quad \neg n(X)$

any computation of a system refining L must lead to state $\{n(1)\}$. Let us now consider rule:

R: **cons** $n(X)$ **ask** $n(Y)$ **ask** $X = Y + 1$ $|$

R satisfies the hypothesis of proposition 5.17, but it does not refine

$S: \; n(X) + n(Y) \wedge X = (Y+1)$ UNLESS $\neg n(X)$,

Consequently, system $\Sigma = (\{R\}, \emptyset, s)$ does not refine L (condition 2 of corollary 5.19 is not satisfied). Indeed, if $n(2)$ is consumed first, then Σ stops in state $\{n(1), n(3)\}$. An alternative rule, refining both L and S is the following:

R': **cons** $n(X)$ **ask** $n(Y)$ **ask** $X = Y + 1$ **ask** $Z > X$ **ask not** $n(Z)$ $|$

5.3 Heterogeneity

Heterogeneity deals with the intermediate steps of a step-wise derivation, the *unrefined programs*, which can be written in the same language of the final program, as in [49], in a specification language, as in [22, 23, 31, 44], or in a mix of the two, as advocated in [1]. In the first case we have the advantage that we can execute and test the unrefined programs, in the second case we can refine the system abstracting from the low level details of a programming language. Finally, in the third, heterogeneous case, we can concentrate on the refinement of a part of the program assuming that the others (usually referred to as the *environment*) behave correctly.

We permit to exploit logic formulae to specify the behaviour of unrefined programs. Each intermediate step is partially written in a tuple space language, and some unrefined features are described by logical formulae. In the heterogeneous view, a system consists of safety properties, liveness properties, rules, a priority order, and an initial state: $(S, L, \mathbb{R}_1^m, P, Is)$. Each of these components can be empty. In the first refinement step $m = 0$, $P = \emptyset$, and $Is = \emptyset$, i.e., we start from a specification, in the last one $L = \emptyset$, i.e., we end up into an executable system. A system $(S, L, \mathbb{R}_1^m, P, Is)$ has to satisfy some constraints, in particular $\mathbb{R}_1^m$ have to preserve the safety constraints in S. Such a condition gives us the ability to compose these systems.

Definition 5.21 ($s(L)$, safe-liveness properties) *Let L be a set of liveness properties, we call $s(L)$ the set:*

$$\{p \text{ UNLESS } q \;\mid\; p \text{ UNTIL } q,\;\; p \mapsto q \;\; or \;\; p \hookrightarrow q \;\in L\}$$

Definition 5.22 (Heterogeneous system) *A system $(S, L, \mathbb{R}_1^m, P, Is)$ satisfies:*

1) $Is \rhd S$

2) $\mathbb{R}_1^m \rhd S \cup s(L)$

In other words, if at a given refinement step, our system is an heterogeneous one, say: $(S, L, I\!R_1^m, P, Is)$, we require not only the *environment* to satisfy properties in S and L, but we also impose some constraints to the rules we already defined. In particular, in order for $(S, L, I\!R_1^m, P, Is)$ to be an heterogeneous system, we require rule $I\!R_1^m$ to refine both the safety properties in S and the safe-liveness properties in $s(L)$.

Definition 5.23 (Computations of heterogeneous systems) *A computation $c = s_0, s_1, \ldots,$ of an heterogeneous system $\Sigma = (S, L, I\!R_1^m, P, Is)$ is a model for the formulae in $S \cup L$, satisfies fairness with respect to the transition system defined by rules $I\!R_1^m$, and satisfies: $Is \subseteq s_0$, (with $\subseteq$ multiset inclusion) and $s_0 \rhd S$. We call $C(\Sigma)$ the set of computations of Σ.*

Definition 5.24 (Systems refinement) *Let Σ and Σ' be systems (either heterogeneous or executable), $\Sigma \rhd \Sigma'$ iff $C(\Sigma) \subseteq C(\Sigma')$.*

The next statement introduces a refinement pattern (refinement by *fulfillment*) for heterogeneous systems, permitting to cancel liveness properties that are fulfilled by the introduced rules, and to substitute them with suitable safety properties. We can read it in the following perspective. An heterogeneous system refines another one if new rules are introduced to "give a help", i.e. to satisfy liveness. In some sense we can look at these new rules as to the *environment* of the rules of the unrefined system.

Proposition 5.25 (Refinement by fulfillment) *Let $(S, L, I\!R_1^m, P, Is)$ be an heterogeneous system, L', L'' liveness sets, such that $L = L' \cup L''$, R be a rule, and let the partial order P' extend P to accommodate R. Then,*

$$(S \cup s(L'), L'', I\!R_1^m \| R, P', Is) \rhd (S, L, I\!R_1^m, P, Is)$$

provided that

$$R \rhd L'$$

$$R \rhd_H S \cup s(L') \cup s(L'')$$

$$(\{R_k \mid R_k \geq R\},\ P',\ Is)\ \rhd\ up(R_j, true)\ \text{UNTIL}\ \neg\, up(R_j, true) \quad \textit{for all } R_j \geq R.$$

with $H = \{\neg\, up(R_k, true) \mid R_k > R \in P'\}$.

Proof. In order for the new system to refine some UNTIL properties, we have to guarantee that the added rule R refines the *progress*, that the new system is an heterogeneous one, and that R, as well as the rules with smaller priority than R, will fire sometimes, i.e. that a state is reached in which $up(R_j, true) = false$, being R_j either R or a rule having greater priority than R. The set of safety properties is augmented, so that further refinements do not interfere with R.

In the special case of empty priority set the proposition above reduces to:

Corollary 5.26 $(S \cup s(L'), L'', I\!R_1^m \| R, \emptyset, Is) \rhd (S, L, I\!R_1^m, \emptyset, Is)$*, provided that:*

$$R \rhd L'$$

$$R \rhd S \cup s(L') \cup s(L'')$$

Working with heterogeneous systems encourages a methodology of refinement: starting from a specification containing *liveness* and *safety* properties, a refinement step is guided by liveness (we introduce rules "doing something good"), and must satisfy safety (the introduced rules "don't have to do anything bad"). When all the liveness properties are guaranteed, we are left with system $(S,\ \emptyset,\ I\!R_1^m,\ P,\ Is)$ where all the safety properties in S are satisfied by $I\!R_1^m$. Proposition 5.29 shows that the executable system $(I\!R_1^m,\ P,\ Is)$ refines it.

The constraints we impose on heterogeneous systems make our approach different from the approaches to *modular refinement* based on a stronger distinction between local system and environment. For instance, in the *rely-guarantee* approach, Jones makes a clear distinction between properties that the local state has to *guarantee* and properties on which we can *rely* [33]. This generates some problems when trying to compose different systems. The topic of composition in this framework is well described by Abadi and Lamport [1].

The following two propositions introduce other refinement patterns: *specification* and *initial state* refinement, respectively.

Proposition 5.27 (Specification refinement) *Let $(S, L, \mathbb{R}_1^m, P, Is)$ be an heterogeneous system, and let S', L' sets of safety and liveness properties, respectively, then $(S', L', \mathbb{R}_1^m, P, Is) \triangleright (S, L, \mathbb{R}_1^m, P, Is)$ if*

$S' \cup L' \to S \cup L,$

$\mathbb{R}_1^m \triangleright (S' \setminus S) \cup s(L' \setminus L),$

$Is \triangleright S' \setminus S$

Proposition 5.28 (Initial state refinement) *Let $(S, L, \mathbb{R}_1^m, P, Is)$ be an heterogeneous system, and let Is' an initial state, then $(S, L, \mathbb{R}_1^m, P, Is') \triangleright (S, L, \mathbb{R}_1^m, P, Is)$ if*

$Is' \supseteq Is$

$Is' \triangleright S$

Proposition 5.29 *Let $\Sigma = (S, \emptyset, \mathbb{R}_1^m, P, Is)$ be an heterogeneous system, and let $\Sigma' = (\mathbb{R}_1^m, P, Is)$ be the corresponding executable system. Then $\Sigma' \triangleright \Sigma$*

Proof. We have to prove that $C(\Sigma') \subseteq C(\Sigma)$ (definition 5.24). Let $c \in C(\Sigma')$, according to definitions 5.2 and 5.5, c satisfies fairness with respect to the transition system defined by rules $\mathbb{R}_1^m$. We know that Σ is an heterogeneous system, hence any computation in the transition system defined by $\mathbb{R}_1^m$ is a model for S, and $Is \triangleright S$. Then, by definition 5.23, $c \in C(\Sigma)$. □

Corollary 5.30 *Let $\Sigma = (\mathbb{R}_1^m, P, Is)$ be an executable system, obtained by refinement steps (applying propositions 5.25, 5.27, 5.28, 5.29) from $(S, L, \emptyset, \emptyset, \emptyset)$, then every computation of Σ refines the liveness properties in L, and preserves the safety conditions in S.*

Finally, we show how to compose heterogeneous systems.

Proposition 5.31 (Composing heterogeneous systems) *Let $(S', L_1' \cup L_2', \emptyset, \emptyset, \emptyset)$ be a specification, $\mathbb{R}_1^m$ and $\mathbb{R}_{1+m}^n$ distinguished sets of rules, and let:*

$(S, L_1, \mathbb{R}_1^m, P_1, Is_1) \triangleright (S' \cup s(L_2'), L_1', \emptyset, \emptyset, \emptyset)$

$(S, L_2, \mathbb{R}_{1+m}^n, P_2, Is_2) \triangleright (S' \cup s(L_1'), L_2', \emptyset, \emptyset, \emptyset)$

$Is_1 \cup Is_2 \triangleright S'$

then

$(S, L_1 \cup L_2, \mathbb{R}_1^n, P_1 \cup P_2, Is_1 \cup Is_2) \triangleright (S', L_1' \cup L_2', \emptyset, \emptyset, \emptyset)$

Proof. The statement directly applies proposition 5.25, (in a simplified version, since here the priority sets are disjoint), and proposition 5.27.

6 An Example: Eratosthenes' sieve

We start with the problem specification, using the following encoding:

$primes(P) :-$ a request to compute primes up to P has been issued.

$prime(X) :-$ X is prime.

We assume $P > 2$.

System E_0 states that all and only prime numbers will be computed:

$$
\begin{aligned}
E_0 : \quad [\, L :&\ \ primes(P)\ \wedge\ 2 \leq X \leq P\ \wedge\ \neg(2 \leq Y < X\ \wedge\ Y|X) \quad \hookrightarrow \quad prime(X) \\
S :&\ \ prime(X)\ \ \text{NEEDS}\ \ primes(P)\ \wedge\ 2 \leq X \leq P\ \wedge\ \neg(2 \leq Y < X\ \wedge\ Y|X)\,]
\end{aligned}
$$

$\lhd$ {we have to split L, to introduce the generator and the sieve. We also have to split safety accordingly. L_1 says that we need consider all the eligible numbers. L_2 reconstructs L, according to C6 (table 5) and

$primes(P)\ \wedge\ 2 \leq X \leq P \quad \hookrightarrow \quad n(X)$

$\Rightarrow$

$primes(P)\ \wedge\ 2 \leq X \leq P\ \wedge\ \neg(2 \leq Y < X\ \wedge\ Y|X) \quad \hookrightarrow \quad n(X)\ \wedge\ \neg(2 \leq Y < X\ \wedge\ Y|X)$

With respect to safety, S_1 states that numbers are generated only for the sieve, S_2 that a prime $prime(X)$ is computed only if X is prime and the corresponding number $n(X)$ has already been generated. Finally S_3 says that the request to compute primes has to be maintained. Refinement correctness for safety is ensured by N6 (table 5) and

$n(X)\ \ \text{NEEDS}\ \ primes(P)\ \wedge\ 2 \leq X \leq P$

$\Rightarrow$

$n(X)\ \wedge\ \neg(2 \leq Y < X\ \wedge\ Y|X)\ \ \text{NEEDS}\ \ primes(P)\ \wedge\ 2 \leq X \leq P\ \wedge\ \neg(2 \leq Y < X\ \wedge\ Y|X)$ }

$$
\begin{aligned}
E_1 : \quad [\, L_1 :&\ \ primes(P)\ \wedge\ 2 \leq X \leq P \quad \hookrightarrow \quad n(X) \\
L_2 :&\ \ n(X)\ \wedge\ \neg(2 \leq Y < X\ \wedge\ Y|X) \quad \mapsto \quad prime(X) \\
S_1 :&\ \ n(X)\ \ \text{NEEDS}\ \ primes(P)\ \wedge\ 2 \leq X \leq P \\
S_2 :&\ \ prime(X)\ \ \text{NEEDS}\ \ n(X)\ \wedge\ \neg(2 \leq Y < X\ \wedge\ Y \mid X) \\
S_3 :&\ \ \text{STABLE}\ \ primes(P)\,]
\end{aligned}
$$

$\lhd$ {we do not want the sieve to explicitly compute primes, but to cancel non-primes: we encode $prime(X)$ as being in the space once "done": $prime(X) \Leftarrow n(X) + done$, where $done$ says that only primes are in the state. We encode with $max(P)$ the condition that all the primes up to P are in the state. At this stage of the refinement, we assume this condition to be angelically written in the state at some point in the computation (L_3).

We have:

$done + primes(P) + max(P) + n(X)\ \wedge\ 2 \leq X \leq P \quad \Rightarrow \quad \neg(2 \leq Y < X\ \wedge\ Y \mid X)$

$done + primes(P) + max(P)\ \wedge\ 2 \leq X \leq P\ \wedge\ \neg(2 \leq Y < X\ \wedge\ Y \mid X) \quad \Rightarrow \quad n(X)$

i.e. all and only primes are in the space. Moreover, we have:

$$max(P) + primes(P) \;\wedge\; 2 \le X \le P \;\wedge\; \neg(2 \le Y < X \;\wedge\; Y \mid X) \;\;\Rightarrow\;\; n(X)$$

while the converse is not true, i.e. we do not force the presence of all the primes up to (P) to imply $max(P)$. We refine S_2 according to this new encoding. S_4 says that no new number can be generated after $max(P)$, S_5 says that primes cannot be lost, and S_6 that $max(P)$ is kept unless *done*.}

E_2 : [L_1 : $primes(P) \wedge 2 \le X \le P \;\hookrightarrow\; n(X)$

L_2 : $n(X) \wedge 2 \le Y < X \le P \wedge Y \not| X \;\mapsto\; n(X) + done$

L_3 : $\Diamond\; max(P)$

S_1, S_3 : =

S_2 : *done* NEEDS $primes(P) + max(P) + \neg(n(X) \wedge 2 \le Y < X \wedge Y \mid X)$

S_4 : $n(X)$ NEEDS $\neg\, max(P)$

S_5 : $n(X)$ UNLESS $2 \le Y < X \wedge Y \mid X$

S_6 : $max(P)$ UNLESS *done*]

$\lhd$ {we refine the sieve by explicitly stating that non primes must be eliminated ($L_{2.1}$), and that *done* must appear once all and only the primes are left ($L_{2.2}$). Refinement correctness is guaranteed by the safety properties and L_3. }

E_3 : [L_1, L_3 : =

$L_{2.1}$: $n(X) \wedge 2 \le Y < X \wedge Y \mid X \;\hookrightarrow\; \neg\, n(X)$

$L_{2.2}$: $primes(P) + max(P) + \neg(n(X) \wedge 2 \le Y < X \wedge Y \mid X) \;\mapsto\; done$

S_1, S_2, S_3, S_4, S_5, S_6 : =]

$\lhd$ {to refine $L_{2.1}$ we introduce rule

R_2: **cons** $n(X)$ **ask** $n(Y)$ **ask** $Y|X$ |

which better captures the expected behaviour of the sieve than the following immediate refinement of $L_{2.1}$:

cons $n(X)$ **ask** $2 \le Y < X$ **ask** $Y|X$ |

A non-prime $n(X)$ is canceled every time a pair $n(X)$, $n(Y)$ is found, with $Y \mid X$. Note that given $n(X)$ and $2 \le Y < X$ with, $Y|X$, it can happen that $n(Y)$ is canceled before $n(X)$, ending up in a situation similar to that of example 5.20. However, in this case we are ensured (S_5) that a $n(Z)$ exists in the space, with Z prime, that can be used to cancel $n(X)$.

We also refine $L_{2.2}$: to avoid the difficulty with the negation, since the premise of $L_{2.2}$ implies the negation of that of $L_{2.1}$, we exploit priorities, introducing

R_3: **ask** $primes(P)$ **ask** $max(P)$ **ask not** *done* | **tell** *done*

and $R_2 > R_3$

We add **ask not** *done* since one is enough. Refinement proofs are in section 6.1. According to proposition 5.25, we have to add the safe-liveness properties

$s(L_{2.1})$: $n(X) \wedge 2 \le Y < X \wedge Y \mid X$ UNLESS $\neg n(X)$

$s(L_{2.2})$: $primes(P) + max(P) + \neg\, done + \neg(n(X) \wedge 2 \le Y < X \wedge Y|X)$ UNLESS *done*

but the first one is a tautology, and the second is implied by S_3, S_6, and S_4. }

$E_4 :\ \ [\, L_1,\ L_3 : =$
$S_1,\ S_2,\ S_3,\ S_4,\ S_5,\ S_6 : =$
$\{R_2,\ R_3\},\ \{R_2 > R_3\}]$

$\lhd$ {we could refine L_1 by introducing rule

R: **ask** $primes(P)$ **ask** $X \leq P$ **ask not** $n(X)$ **ask not** $max(P)$ $|$ **tell** $n(X)$

but it would be highly inefficient, since non primes would be continuously restored. Besides, this rule would not "implement" the angelic behaviour that we have assumed with respect to max(P). We hence refine E_4 in order to generate all the numbers orderly and once, according to the following specification refinement:

$L_1 :\ primes(P) \wedge 2 \leq X \leq P \quad \hookrightarrow \quad n(X)$

$\Leftarrow$

$L_{1.1} :\ primes(P) + max(M) \wedge M < P \ \ \textsc{until} \ \ max(M+1)$

$S_7 :\ max(M) \ \ \textsc{needs} \ \ n(M) \wedge 2 \leq M \leq P$

$S_8 :\ \Box \neg (n(X) + n(X))$

$S_9 :\ \textsc{init} \ \ max(2)$

Moreover, since $\Diamond\ max(P)$ follows from $L_{1.1}$, also L_3 is refined by $L_{1.1}$}

$E_5 :\ \ [\, L_{1.1} :\ primes(P) + max(M) \wedge M < P \ \ \textsc{until} \ \ max(M+1)$
$S_1,\ S_2,\ S_3,\ S_4,\ S_5,\ S_6 : =$
$S_7 :\ max(M) \ \ \textsc{needs} \ \ n(M) \wedge 2 \leq M \leq P$
$S_8 :\ \Box \neg (n(X) + n(X))$
$S_9 :\ \textsc{init} \ \ max(2)$
$\{R_2,\ R_3\},\ \{R_2 > R_3\}]$

$\lhd$ {we can now refine $L_{1.1}$ by introducing the last rule (see also section 6.2)

R_1: **cons** $max(M)$ **ask** $primes(P)$ **ask** $M < P$
  $|$ **body** $M'\ is\ M + 1$
  tell $max(M')$ **tell** $n(M')$ }

$E_6 :\ \ [\, S_1,\ S_2,\ S_3,\ S_4,\ S_5,\ S_6,\ S_7,\ S_8,\ S_9 : =$
$s(L_{1.1}) :\ primes(P) + max(M) \wedge M < P \ \ \textsc{unless} \ \ max(M+1)$
$\{R_1,\ R_2,\ R_3\},\ \{R_2 > R_3\}\,]$

$\lhd$ {to fix the initial state, we need an initial state refinement, and apply proposition 5.28, with $Is_E = \{max(2), n(2), primes(P)\}$, with $P \geq 2$}

$E_7 :\ \ [\, S_1,\ S_2,\ S_3,\ S_4,\ S_5,\ S_6,\ S_7,\ S_8,\ S_9,\ s(L_{1.1}) : =$
$\{R_1,\ R_2,\ R_3\},\ \{R_2 > R_3\},\ Is_E\,]$

$\lhd$ {we can finally simplify R_3 exploiting priorities:

R_3: **ask not** *done* | **tell** *done*

and $R_1 > R_3$

}

E_7 : [S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, $s(L_{1.1})$: =
$\{R_1,\ R_2,\ R_3\}$, $\{R_1 > R_3,\ R_2 > R_3\}$, $\{max(2),\ n(2)\}$]

$\lhd$ {we can now end up in the executable system of table 2, by proposition 5.29 }

Σ : [$\{R_1,\ R_2,\ R_3\}$, $\{R_1 > R_3,\ R_2 > R_3\}$, $\{max(2),\ n(2)\}$]

6.1 Proof of the fourth refinement step: $E_3 \lhd E_4$

To prove that $E_3 \lhd E_4$, we apply twice proposition 5.25, introducing first R_2 and then R_3. To prove that the introduction of R_2 to fulfill $L_{2.1}$ is correct, we have to prove that R_2 refines: $L_{2.1}$, $s(L_1)$, the safety properties in E_4, and $up(R_2, true)$ UNTIL $\neg\, up(R_2, true)$.

Similarly, to prove that the introduction of R_3 is correct, we have to prove that R_3 refines $L_{2.2}$ and $up(R_3, true)$ UNTIL $\neg\, up(R_3, true)$, and that it refines $s(L_1)$ and the safety properties in E_4 under the hypothesis $H_3 = \{\neg\, up(R_2, true)\}$ (proposition 5.14).

We show a couple of them.

$R_2 \rhd L_{2.1}$ (we apply proposition 5.17)

In table 7 we exploit S_8 and prove that:

$(n(X) + n(Y) \wedge Y \mid X) \rightarrow ep(R_2, \neg n(X))$

we now thus left with:

$n(X) \ \wedge\ 2 \leq Y < X \ \wedge\ Y \mid X \quad \mapsto \quad n(X) + n(Z) \wedge Z \mid X$

which exploits S_5 (primes are left in the space) and conclude applying C6 of table 5.
□

$R_3 \rhd L_{2.2}$ To apply proposition 5.17, we have to show that

$$primes(P) + max(P) + \neg\,(n(X) + n(Y) \ \wedge\ Y \mid X) + \neg\, done \quad \rightarrow \quad ep(R_3, done)$$

which is trivial, since calculation show that $ep(R_3, done)$ coincides with the premise.
□

6.2 Proof of the sixth refinement step: $E_5 \lhd E_6$

To prove $E_5 \lhd E_6$, we apply again proposition 5.25. We have to prove that R_1 refines: $L_{1.1}$, the safety properties in E_6, and $up(R_1, true)$ UNTIL $\neg\, up(R_1, true)$; also, we have to prove that R_2 and R_3 refine $s(L_{1.1})$. We show some of them.

$R_1 \rhd L_{1.1}$ To apply proposition 5.17) we have to show that

$$primes(P) + max(M) \wedge M < P \quad \rightarrow \quad ep(R_1, max(N+1))$$

which is trivial, since

$ep(R_1, max(M+1)) = primes(P) + max(M) \wedge M < P$ $\square$

$R_1 \rhd S_1$ To apply proposition 5.10 we have to show that

$$ep(R_1, n(X) \wedge \neg(primes(P) \wedge 2 \leq X \leq P)) \quad \rightarrow \quad n(X)$$

which is also trivial, since

$ep(R_1, n(X) \wedge \neg(primes(P) \wedge 2 \leq X \leq P)) = n(X) + primes(P) + max(M) \wedge M < P$
$\square$

$R_1 \rhd S_2$ To apply proposition 5.10 we have to show that

$$ep(R_1, done \wedge [\neg primes(P) \vee \neg max(P) \vee (n(X) \ \wedge\ 2 \leq Y < X \ \wedge\ Y|X]) \ \rightarrow\ done$$

which is true, by proposition 4.15, since

$ep(R_1, done) \rightarrow \ done$ $\square$

$R_1 \rhd S_5$ We exploit the fact that R_1 does not consider $n(\_)$, and we prove the stronger $R_1 \rhd n(X)$ UNLESS $false$, obtaining a shorter proof. To apply proposition 5.9, under the stronger assumption, we have to prove that

$$n(X) \quad \rightarrow \quad (up(R_1, n(X)) \vee \neg up(R_1, true))$$

Indeed, $up(R_1, n(X)) \vee \neg up(R_1, true)$

$$= primes(P) + max(M) \ \wedge\ M < P \ \wedge\ [(n(X) \ \wedge\ M \neq X-1) \ \vee\ M = X-1]$$
$$\vee \ \neg(primes(P) + max(M) \ \wedge\ M < P)$$

and this follows from $n(X)$, by simple propositional calculations. $\square$

6.3 Proof of the eighth refinement step: $E_7 \lhd E_8$

We show the most interesting proof, i.e. that S_2 is still refined.

$R_3 \rhd_{H_3} S_2$

To apply proposition 5.13), we have to show that

$ep(R_3, done + \mathcal{C}) \rightarrow \ done \ \vee \ \neg H_3$

where $H_3 = \neg up(R_1, true) \ \wedge\ \neg up(R_2, true)$, according to proposition 5.14.

and $\mathcal{C} = \neg primes(P) \vee \neg max(P) \vee (n(X) \ \wedge\ 2 \leq Y < X \ \wedge\ Y|X)$

Since

$ep(R_3, done + \mathcal{C}) \ = \ \neg done + \mathcal{C}$

we have only to show that $\neg done + \mathcal{C} \ \rightarrow \ \neg H_3$. Then:

$\neg H_3$

$$
\begin{array}{ll}
= & wp(R_1, true) \ \vee \ wp(R_2, true) \\
= & wp(R_1, true) \ \vee \ (n(X) + n(Z) \ \wedge \ Z|X) \\
 & \{S_5\} \\
= & wp(R_1, true) \ \vee \ (n(X) \ \wedge \ 2 \leq Y < X \ \wedge \ Y|X)] \\
= & (primes(P) + max(M) \ \wedge \ M < P) \ \vee \ (n(X) \ \wedge \ 2 \leq Y < X \ \wedge \ Y|X)] \\
 & \{S_3, \ S_9, \ s(L_{1.1})\} \\
\leftarrow & \neg\, done + [\neg\, primes(P) \vee \neg\, max(P) \vee (n(X) \ \wedge \ 2 \leq Y < X \ \wedge \ Y|X)] \\
= & \neg\, done + \mathcal{C}
\end{array}
\qquad \square
$$

Conclusions

Tuple space languages are an acknowledged model to describe cooperation and, more generally, to program large distributed reactive systems.

To build a refinement calculus for this framework, we have defined: a variation of temporal logic to provide readable and concise specifications; an axiomatic semantics to derive properties of a tuple space system; an operational semantics to describe the allowed computations. The calculus is then based on the relation between these descriptions.

One of the contributions of our work is the idea to derive weakest preconditions exploiting Back's *demonic strict* choice in non-deterministic selection. The transition system defining the operational semantics is based on the new notion of enabling precondition, which exploits the *angelic strict* choice. We use weakest and enabling preconditions to define the refinement of safety and liveness properties, respectively.

The calculus has been exploited to model and implement software processes, to describe software architectures, and may be extended to cope with software maintenance. In [40] we provide evidence that the advantages of a refinement calculus are gained also for those programs known as *enactable process models*. The evidence is put forward by means of an example, a small Concurrent Engineering problem inspired by the ISPW-7 problem. In [41], we illustrate some refinement patterns within the tuple space framework. Since each pattern introduces an architectural style, they are called *architectural refinements*. We instantiate them in the refinement of a particular system: a fragment of a software process. Then, the refinement calculus presented here provides a firm formal basis for a method that besides correctness, highlights the architectural choices in the design.

Acknowledgments

This paper presents and extends part of the results stated in the first author's thesis. We gratefully thank Krzysztof Apt, Dino Pedreschi, Gruia-Catalin Roman, and Franco Turini, who refereed the thesis, providing many useful suggestions.

References

[1] M. Abadi and L. Lamport. Composing Specifications. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Proc. Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 1–41. Springer-Verlag, 1989.

[2] B. Alpern and F. Shneider. Defining Liveness. *Inf. Proc. Letters*, 21(4):181–185, 1985.

[3] V. Ambriola, P. Ciancarini, and C. Montangero. Software Process Enactment in Oikos. In R. Taylor, editor, *Proc. ACM SIGSOFT 90*, 4^{th} *Symposium on Software Development Environments*, pages 183–192, Irvine, Ca., 1990.

[4] V. Ambriola, G. A. Cignoni, and L. Semini. eta - Everything buT Assignment. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proc. of the 1994 Joint Conference on Declarative Programming GULP-PRODE'94*, Peniscula, Sept 1994.

[5] V. Ambriola, G. A. Cignoni, and L. Semini. A Proposal to Merge Object Orientation, Logic Programming, and Multiple Tuple Spaces. *Computer Languages*, 22(2/3):79–93, 1996.

[6] V. Ambriola and L. Semini. Control Specification in Tuple Space Based Languages. Technical Report 19-93, Dipartimento di Informatica, Università di Pisa, 1993. See also pate.dvi in [50].

[7] J. M. Andreoli and R. Pareschi. Linear Objects: logical processes with built-in inheritance. In D. H. D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming*, pages 495–510. The MIT Press, Cambridge, Mass., 1990.

[8] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, Berlin, 1991.

[9] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Pratice and Experience*, 5(1):23–70, Feb 1993.

[10] R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[11] R. Back and J. von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Proc. Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1989.

[12] J. Banatre and D. L. Métaier. The Gamma model of computation and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[13] J. Banatre and D. L. Métaier. Programming by Multiset Transformation. *Communications of the ACM*, 16(1):55–77, 1993.

[14] A. Brogi and P. Ciancarini. The Concurrent Language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[15] M. Broy. Functional Specification of Time Sensitive Communicating Systems. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Proc. Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 153–179. Springer-Verlag, 1989.

[16] M. Broy. (Inter-) Action Refinement: The Easy Way. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series, pages 121–158. Springer-Verlag, Berlin, 1993.

[17] A. Bucci, P. Ciancarini, and C. Montangero. A Distributed Logic Language Based on Multiple Tuple Spaces. In *Proc. Logic Programming Conference*, pages 199–208, Tokio, 1991.

[18] N. Carriero and D. Gelernter. Linda and Friends. *IEEE Computer*, 19(8):26–34, 1986.

[19] N. Carriero and D. Gelernter. Coordination Languages and their Significance. *Communications of the ACM*, 5(2):97–107, 1989.

[20] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–459, 1989.

[21] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading Mass., 1988.

[22] X. J. Chen. *Formalism and Verification towards Software Process Modelling.* PhD thesis, Scuola Normale Superiore di Pisa, 1992.

[23] X. J. Chen and C. Montangero. Compositional Refinements of Multiple Blackboard Systems. *Acta Informatica*, 32(5):415–458, 1995.

[24] P. Ciancarini. PoliS: a Programming Model for Multiple Tuple Spaces. In *Proc.* 6^{th} *IEEE International Workshop on Software Specification and Design*, 1991.

[25] P. Ciancarini and C. Hankin, editors. *Proc. COORDINATION 96*, volume 1061 of *Lecture Notes in Computer Science*, Cesena, April 1996. Springer-Verlag.

[26] H. C. Cunningam and G.-C. Roman. A UNITY-Style Programming Logic for Shared Dataspace Programs. *IEEE Transaction on Parallel and Distributed Systems*, 1(3):365–376, Jul 1990.

[27] E. Dijkstra and C. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, Berlin, 1990.

[28] A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology.* Research Study Press distributed by J. Wiley & sons, London, 1994.

[29] M. Gaspari and L. Semini. Fairness in Logic Languages based on Shared Dataspace. In K. D. Bosshere, J. Jacquet, and P. Tarau, editors, *Proc. of the ICLP'93 Post-Conference Workshop on Blackboard-Based Logic Programming*, Budapest, 1993. See also iclp93.dvi in [50].

[30] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[31] C. Hoare. Programs are Predicates. In C. Hoare and J. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 141–155. Prentice-Hall, 1985.

[32] C. Hoare. Algebra and Models. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series, pages 161–195. Springer-Verlag, Berlin, 1993.

[33] C. Jones. Specification and Design of (Parallel) Programs. In R. Mason, editor, *Information Processing 83*, pages 321–332. Norh-Holland, 1983.

[34] F. Kröger. *Temporal Logic of Programs.* Springer-Verlag, Berlin, 1987.

[35] L. Lamport. What it Means for a Concurrent Program to Satisfy a Specification: Why no One has Specified Priority. In *Proc. 12th Annual ACM Symp. on Principles of Programming Languages*, pages 79–83, New Orleans, 1985.

[36] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987.

[37] Z. Manna and A. Pnueli. Verification of Concurrent Programs: A Temporal Proof System. In J. de Bakker and J. van Leeuwen, editors, *Foundations of Computer Science IV, Distributed Systems: Part 2*, Mathematical Centre Tracts 159, pages 163–255, Amsterdam, 1983.

[38] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, New-York, 1992.

[39] C. Montangero and V. Ambriola. Oikos: Constructing Process-centered SDEs. pages 131–151. In [28].

[40] C. Montangero and L. Semini. Applying Refinement Calculi to Software Process Modelling. In W. Schaefer, editor, *Proc 4th International Conference on the Software Process ICSP4*, pages 63–74, Brighton, UK, Dec 1996. IEEE Computer Society Press, Los Alamitos. Preliminary version available as TR-96-18, Dipartimento di Informatica, May 22, 1996, at http://www.di.unipi.it/cgi-bin/masterform.

[41] C. Montangero and L. Semini. Refining by Architectural Styles. In L.Vidal, A.Finkelstein, G. Spanoudakis, and A. Wolf, editors, *2nd International Software Architecture Workshop, Joint Proceedings of the SIGSOFT '96 Workshop*, pages 76–79, San Francisco, CA, Oct 1996.

[42] L. Monteiro and A. Porto. Entailment Based Actions for Coordination. *Theoretical Computer Science*, Forthcoming.

[43] C. Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[44] C. Morgan. The Refinement Calculus. In M. Broy, editor, *Program Design Calculi*, NATO ASI Series, pages 3–52. Springer-Verlag, Berlin, 1993.

[45] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, Berlin, 1986.

[46] A. Porto and P. Rosado. The AbstrAct scheme for concurrent programming. In E. Lamma and P. Mello, editors, *Extensions of Logic Programming*, volume 660 of *LNAI*, pages 216–241. Springer-Verlag, 1993.

[47] A. Porto and V. Vasconcelos. Truth and action osmosis (the tao computation model). In J. Andreoli, C. Hankin, and D. LeMetayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*, pages 65–97. Imperial College Press, 1996.

[48] G. C. Roman and H. C. Cunningam. Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency. *IEEE Transactions on Software Engineering*, 16(2):1361–1376, 1990.

[49] G. C. Roman, R. F. Gamble, and W. E. Ball. Formal Derivation of Rule-Based Programs. *IEEE Transactions on Software Engineering*, 19(3):277–296, March 1993.

[50] L. Semini. ftp personal directory. ftp.di.unipi.it/pub/Papers/semini/.

[51] L. Semini. *Refinement in Tuple Space Languages.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1996. TD 10-96. At www.di.unipi.it/~semini/semini.html.

[52] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Mass., second edition, 1994.

A Appendix: Proofs from sections 3 and 4

Proposition 3.10 (pag. 14) $\sim p(\bar{\mathrm{x}}) \;\equiv\; \neg\, p(\bar{\mathrm{x}})$

Proof Let $\mathcal{M} = \langle S, V\rangle$ be a model, and s a state in S.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{M},\, s \Vdash t(\sim p(\bar{\mathrm{x}}))} \\
\Leftrightarrow & \mathcal{M},\, s \Vdash \exists\vartheta_{p_{\setminus ex}} (\sim p(\bar{\mathrm{x}}), \vartheta) \\
\Leftrightarrow & \exists\vartheta_{p_{\setminus ex}}\;\; \mathcal{M},\, s \Vdash (\sim p(\bar{\mathrm{x}}), \vartheta) \\
\Leftrightarrow & \exists\vartheta_{p_{\setminus ex}}\;\; (\sim p(\bar{\mathrm{x}}), \vartheta) \in V(s) \\
\Leftrightarrow & \exists\vartheta_{p_{\setminus ex}}\;\; s \models (\sim p(\bar{\mathrm{x}}), \vartheta) \\
\Leftrightarrow & \exists\vartheta_{p_{\setminus ex}}\;\; s \models_w (\sim p(\bar{\mathrm{x}}), \vartheta) \text{ with } \vartheta \text{ success subst. for } \sim p(\bar{\mathrm{x}}) \text{ and } s \\
\Leftrightarrow & s \models_w (\sim p(\bar{\mathrm{x}}), \varepsilon) \\
\Leftrightarrow & \forall\vartheta_p \; \textit{not}\; s \models_w (p(\bar{\mathrm{x}}), \vartheta \circ \varepsilon) \\
\Leftrightarrow & \forall\vartheta_p \; \textit{not}\; s \models_w (p(\bar{\mathrm{x}}), \vartheta) \text{ with } \vartheta \text{ success subst. for } p(\bar{\mathrm{x}}) \text{ and } s \\
\Leftrightarrow & \forall\vartheta_p \; \textit{not}\; s \models (p(\bar{\mathrm{x}}), \vartheta) \\
\Leftrightarrow & \forall\vartheta_p \; \textit{not}\; (p(\bar{\mathrm{x}}), \vartheta) \in V(s) \\
\Leftrightarrow & \forall\vartheta_p \; \textit{not}\; \mathcal{M},\, s \Vdash (p(\bar{\mathrm{x}}), \vartheta) \\
\Leftrightarrow & \forall\vartheta_p \; \mathcal{M},\, s \Vdash \neg\,(p(\bar{\mathrm{x}}), \vartheta) \\
\Leftrightarrow & \mathcal{M},\, s \Vdash \forall\vartheta_p \, \neg\,(p(\bar{\mathrm{x}}), \vartheta) \\
\Leftrightarrow & \mathcal{M},\, s \Vdash t(\neg\, p(\bar{\mathrm{x}}))
\end{array}
$$

□

Proposition 4.9 (pag. 24) Let p and q be state formulae and R a rule. Then

$$wp(R,\, p \wedge q) \equiv wp(R,\, p) \wedge wp(R,\, q)$$

Proof We apply definition 4.7 (pag. 21) and show:

1. $wp(\textsc{tell},\, p \wedge q) \equiv wp(\textsc{tell},\, p) \wedge wp(\textsc{tell},\, q)$
2. $wp(\textsc{body}\;,\, \{p_1 \wedge q_1,\, p_2 \wedge q_2\}) \equiv wp(\textsc{body}\;,\, \{p_1,\, p_2\}) \wedge wp(\textsc{body}\;,\, \{q_1,\, q_2\})$
3. $wp(\textsc{cons}\;,\, p \wedge q) \equiv wp(\textsc{cons}\;,\, p) \wedge wp(\textsc{cons}\;,\, q)$
4. $wp(\textsc{guard},\, p \wedge q) \equiv wp(\textsc{guard}, p) \wedge wp(\textsc{guard}, p)$

1. For every $\textbf{tell}\; t(\bar{\mathrm{B}})$ in $\textsc{tell}$:

$$
\begin{array}{ll}
wp(\textbf{tell}\; t(\bar{\mathrm{B}}),\, p \wedge q) \;\; \equiv & \\
ut(p \wedge q, t(\bar{\mathrm{B}}))\, [t(\bar{\mathrm{B}})^{\bullet\, n} / t(\bar{\mathrm{B}})^{\bullet\, n-1}] \;\; \equiv & \text{(definition 4.3, pag. 20)} \\
(ut(p, t(\bar{\mathrm{B}})) \wedge ut(q, t(\bar{\mathrm{B}})))\, [t(\bar{\mathrm{B}})^{\bullet\, n} / t(\bar{\mathrm{B}})^{\bullet\, n-1}] \;\; \equiv & \\
ut(p, t(\bar{\mathrm{B}}))\, [t(\bar{\mathrm{B}})^{\bullet\, n} / t(\bar{\mathrm{B}})^{\bullet\, n-1}] \wedge ut(q, t(\bar{\mathrm{B}}))\, [t(\bar{\mathrm{B}})^{\bullet\, n} / t(\bar{\mathrm{B}})^{\bullet\, n-1}] \;\; \equiv & \\
wp(\textbf{tell}\; t(\bar{\mathrm{B}}),\, p) \wedge wp(\textbf{tell}\; t(\bar{\mathrm{B}}),\, q) &
\end{array}
$$

2. $wp(\textbf{body } t(\bar{\text{B}}),\ \{p_1 \wedge q_1,\, p_2 \wedge q_2\}) \ \equiv$

   $[t(\bar{\text{B}}) \ \wedge \ p_1 \wedge q_1] \ \vee \ [\neg\, t(\bar{\text{B}}) \ \wedge \ p_2 \wedge \ q_2)] \ \equiv$

   $\{[t(\bar{\text{B}}) \ \wedge \ p_1] \ \vee \ [\neg\, t(\bar{\text{B}}) \ \wedge \ p_2]\} \ \wedge \ \{[t(\bar{\text{B}}) \ \wedge \ q_1] \ \vee \ [\neg\, t(\bar{\text{B}}) \ \wedge \ q_2]\} \equiv$

   $wp(\textbf{body } t(\bar{\text{B}}),\ \{p_1,\, p_2\}) \wedge wp(\textbf{body } t(\bar{\text{B}}),\ \{q_1,\, q_2\})$

3. For every **cons** $t(\bar{\text{B}})$ in TELL, $wp(\textbf{cons } t(\bar{\text{B}}),\ p \wedge q) \equiv wp(\textbf{cons } t(\bar{\text{B}}),\ p) \wedge wp(\textbf{cons } t(\bar{\text{B}}),\ q)$ is shown in a way analogous to *1.*

4. $wp(\text{GUARD},\, p \wedge q) \ \equiv$

   $g2f(\text{GUARD}) \ \wedge \ (g2f(\text{GUARD}) \ \rightarrow \ p \wedge q) \ \equiv$

   $g2f(\text{GUARD}) \ \wedge \ [g2f(\text{GUARD}) \ \rightarrow \ p] \ \wedge \ [g2f(\text{GUARD}) \ \rightarrow \ q] \ \equiv$

   $wp(\text{GUARD}, p) \wedge wp(\text{GUARD}, p)$ □

Proposition 4.15 (pag. 26) Let p and q be state formulae and R a rule, then :

$$ep(R,\, p \vee q) \equiv ep(R,\, p) \vee ep(R,\, q)$$

Proof The proof structure is similar to the previous one.

1. For every **tell** $t(\bar{\text{B}})$ in TELL (similar to TELL in proposition 4.9):

   $wp(\textbf{tell } t(\bar{\text{B}}),\ p \vee q) \equiv wp(\textbf{tell } t(\bar{\text{B}}\ p) \vee wp(\textbf{tell } t(\bar{\text{B}}\ q)$

2. $wp(\textbf{body } t(\bar{\text{A}}, \bar{\text{B}}),\ \{p_1 \vee q_1,\, p_2 \vee q_2\}) \equiv$

   $[t(\bar{\text{B}}) \ \wedge \ (p_1 \vee q_1)] \ \vee \ [\neg\, t(\bar{\text{B}}) \ \wedge (\ p_2 \vee \ q_2)] \equiv$

   $[t(\bar{\text{B}}) \ \wedge \ p_1] \ \vee \ [t(\bar{\text{B}}) \ \wedge \ q_1] \ \vee \ [\neg\, t\bar{\text{B}}) \ \wedge \ p_2] \ \vee \ [\neg\, t(\bar{\text{B}}) \ \wedge \ q_2] \equiv$

   $[t(\bar{\text{A}}, \bar{\text{B}}) \ \wedge \ p_1] \ \vee \ [\neg\, t(\bar{\text{B}}) \ \wedge \ p_2] \ \vee \ [t(\bar{\text{B}}) \ \wedge \ q_1] \ \vee \ [\neg\, t(\bar{\text{B}}) \ \wedge \ q_2] \equiv$

   $wp(\textbf{body } t(\bar{\text{A}}, \bar{\text{B}}),\ \{p_1,\, p_2\}) \ \vee \ wp(\textbf{body } t(\bar{\text{A}}, \bar{\text{B}}),\ \{q_1,\, q_2\})$

3. For every **cons** $t(\bar{\text{B}})$ we have:

   $wp(\textbf{cons } t(\bar{\text{B}}),\ p \vee q) \equiv wp(\textbf{cons } t(\bar{\text{B}}),\ p) \vee wp(\textbf{cons } t(\bar{\text{B}}),\ q)$

4. $ep(\text{GUARD},\, p \vee q) \equiv$

   $g2f(\text{GUARD}) \ \wedge \ (p \vee q) \equiv$

   $[g2f(\text{GUARD}) \ \wedge \ p] \ \vee \ [g2f(\text{GUARD}) \ \wedge \ q] \equiv$

   $ep(\text{GUARD}, p) \vee ep(\text{GUARD}, p)$ □

Corollary 4.16 (pag. 26) Let p be a state formula and R a rule, then :

$$\neg\, ep(R,\, p) \wedge ep(R,\, true) \ \ \rightarrow \ \ ep(R,\, \neg\, p)$$

Proof $\neg\, ep(R,\, p) \wedge \neg\, ep(R,\, \neg\, p) \wedge ep(R,\, true) \equiv$

$\neg\, ep(R,\, p) \wedge \neg\, ep(R,\, \neg\, p) \wedge ep(R,\, p \vee \neg\, p) \equiv$

$\neg\, ep(R,\, p) \wedge \neg\, ep(R,\, \neg\, p) \wedge [ep(R,\, p) \vee ep(R,\, \neg\, p)] \equiv$

$false$ □

Proposition 4.19 (pag. 27) Let p be a state formula and R a rule, then :

$$wp(R,\, q) \to ep(R,\, q)$$

Proof $wp(\textsc{guard},\, q) \equiv$

$g2f(\textsc{guard}) \;\wedge\; (g2f(\textsc{guard}) \;\to\; q) \;\to$

$g2f(\textsc{guard}) \;\wedge\; q \;\equiv$

$ep(\textsc{guard}, q)$ □

Lemma 4.20 (pag. 27) Let R be a rule, then $wp(R, true) = ep(R, true)$

Proof Let $\textsc{g}$ denote $g2f(\textsc{guard})$.

$wp(R, true) \quad =$ (definition 4.7, pag. 21)

$\textsc{g} \;\wedge\; [\textsc{g} \to true] \quad =$

$\textsc{g} \quad =$

$\textsc{g} \wedge true \quad =$ (definition 4.13, pag. 25)

$ep(R, true)$

We recall that rule R can include a body only one postcondition only if the body cannot fail. □

Proposition 4.21 (pag. 27) Let R be a rule and q a state formula, then

$$wp(R, q) \equiv \neg ep(R, \neg q) \wedge ep(R, true)$$

Proof The proof structure is similar to the one of proposition 4.9.

1. For every **tell** $t(\bar{\textsc{b}})$ in $\textsc{tell}$:

$wp(\textbf{tell}\ t(\bar{\textsc{b}}),\ \neg q) \;\equiv\; ut(\neg q, t(\bar{\textsc{b}}))\,[t(\bar{\textsc{b}})^{\bullet\, n}/t(\bar{\textsc{b}})^{\bullet\, n-1}] \;\equiv$

$\neg ut(q, t(\bar{\textsc{b}}))\,[t(\bar{\textsc{b}})^{\bullet\, n}/t(\bar{\textsc{b}})^{\bullet\, n-1}] \;\equiv\; \neg wp(\textbf{tell}\ t(\bar{\textsc{b}}),\ q)$

2. $wp(\textbf{body}\ t(\bar{\textsc{b}}),\ \{\neg q_1,\, \neg q_2\}) \;\equiv\; [t(\bar{\textsc{b}}) \;\wedge\; \neg q_1] \;\vee\; [\neg\, t(\bar{\textsc{b}}) \;\wedge\; \neg q_2)] \;\equiv$

$[\neg\, t(\bar{\textsc{b}}) \;\vee\; \neg\, q_1] \;\wedge\; [t(\bar{\textsc{b}}) \;\vee\; \neg\, q_2]\} \;\equiv\; \neg\, [t(\bar{\textsc{b}}) \;\wedge\; q_1] \;\wedge\; \neg\, [\neg\, t(\bar{\textsc{b}}) \;\wedge\; q_2]\} \;\equiv$

$\neg\, wp(\textbf{body}\ t(\bar{\textsc{b}}),\ \{q_1,\, q_2\})$

3. For every **cons** $t(\bar{\textsc{b}})$ in $\textsc{tell}$, $wp(\textbf{cons}\ t(\bar{\textsc{b}}),\ \neg\, q) \equiv \neg\, wp(\textbf{cons}\ t(\bar{\textsc{b}}),\ q)$ is shown in a way analogous to *1.*

4. $ep(\textsc{guard},\, true) \;\wedge\; \neg\, ep(\textsc{guard},\, \neg q) \;\equiv$

$g2f(\textsc{guard}) \;\wedge\; \neg\, [g2f(\textsc{guard}) \;\wedge\; \neg\, q) \;\equiv$

$g2f(\textsc{guard}) \;\wedge\; [g2f(\textsc{guard}) \to q] \;\equiv$

$wp(\textsc{guard}, q)$ □

Proposition 4.22 (pag. 27) Let s be a state and R a rule, then:

$$s|\models wp(R, true) \quad \leftrightarrow \quad T(R)(s) \neq \emptyset$$

Proof First implication: $T(R)(s) \neq \emptyset \ \rightarrow \ s|\models wp(R, true)$

$T(R)(s) \neq \emptyset \ \rightarrow$ ($s'|\models true$ and definition 4.17, pag. 27)

$s|\models ep(R, true) \ \leftrightarrow$ (lemma 4.20, pag. 27)

$s|\models wp(R, true)$

We now prove the other implication: $s|\models wp(R, true) \rightarrow T(R)(s) \neq \emptyset$.

Let t be maximal for the property $s|\models wp(R, t)$. Since $t \neq false$, then $\exists s'.\, s'|\models t$.

We show that: $\forall p.[(s'|\models p \wedge s|\not\models ep(R,p)) \rightarrow (s|\not\models wp(R, p) \vee s|\not\models wp(R, t \wedge \neg p))]$

a) $s|\not\models ep(R,p) \wedge s|\models wp(R, p) \overset{\text{prop.4.19}}{\rightarrow} s|\not\models ep(R,p) \wedge s|\models ep(R, p) \ \equiv \ false$

b) $s'|\models p \wedge s|\models wp(R, t \wedge \neg p) \overset{\text{maxim. of } t}{\rightarrow} s'|\models p \wedge s'|\models \neg p \ \equiv \ false$

Let us consider now the formula $t \wedge \wedge_i \neg p_i$ such that

$\exists s_i.s_i|\models (t \wedge p_i) \wedge s|\not\models wp(R, p_i) \wedge s|\not\models wp(R, t \wedge \neg p_i)$

we show by induction that $t \wedge \wedge_i \neg p_i \neq false$

base: let $t \wedge \neg p_1 \equiv false$, then $t \rightarrow p_1$, then (corollary 4.10, pag. 24) $s|\models wp(R, p_1)$, and leads to a contradiction.

inductive step: $t \wedge \wedge_{i=1,\ldots n} \neg p_i \neq false$,

then $\exists s_n.s_n|\models (t \wedge \wedge_{i=1,\ldots n} \neg p_i)$

$\wedge s|\not\models wp(R, \vee_{i=1,\ldots n} p_i) \wedge s|\not\models wp(R, t \wedge \wedge_{i=1,\ldots n} \neg p_i))$

let us assume: $t \wedge \wedge_{i=1,\ldots n} \neg p_i \wedge \neg\, p_{n+1} = false$, then $(t \rightarrow \vee_{1,\ldots n} p_i) \vee (t \rightarrow p_{n+1})$

which implies, for the maximality of t:

$s|\models wp(R, \vee_{i=1,\ldots n} p_i) \quad \vee \quad s|\models wp(R, p_{n+1})$ while both of them are false.

Now, since $t \wedge \wedge_i \neg p_i \neq false$, it means that there exists a state $\bar{s}'$ satisfying it. $\forall r$:

$\bar{s}'|\models r \ \rightarrow$

$\bar{s}'|\models t \wedge \wedge_i \neg p_i \wedge r \ \rightarrow$

$[s|\models wp(R, \wedge_i \neg p_i \wedge r) \ \vee \ s|\models wp(R, t \wedge \neg\, (\wedge_i \neg p_i \wedge r))]$ (A)

Let us assume $s|\not\models ep(R, r)$, we show that this leads to an absurd:

$s|\not\models ep(R, r) \ \rightarrow$ (proposition 4.19, pag. 27)

$s|\not\models wp(R, r) \ \rightarrow$ (proposition 4.9, pag. 24)

$s|\not\models wp(R, \wedge_i \neg p_i \wedge r) \ \rightarrow$ (see A)

$s|\models wp(R, t \wedge \neg\, (\wedge_i \neg p_i \wedge r)) \ \rightarrow$ (maximality of t)

$(t \rightarrow \neg\, (\wedge_i \neg p_i \wedge r)) \ \rightarrow$ ($\bar{s}'|\models t$)

$\bar{s}'|\models \neg\, (\wedge_i \neg p_i \wedge r) \ \rightarrow$

$\bar{s}'|\models \neg\, (\wedge_i \neg p_i) \quad$ or $\quad \bar{s}'|\models \neg\, r$ (construction of $\bar{s}'$ and hypothesis $\bar{s}'|\models r$)

$false$ $\square$

Theorems 4.25 and 4.24 (pag. 27) Let R be a rule, s a state, and q a state formula, then $\forall \bar{\text{Y}}$ free in q:

$$s \| \models wp(R,\, q(\bar{\text{Y}})) \quad \leftrightarrow \quad T(R)(s) \neq \emptyset \wedge \; \forall s'.[s' \in T(R)(s) \rightarrow s' \| \models q(\bar{\text{Y}})]$$

$$s \| \models ep(R,\, q(\bar{\text{Y}})) \quad \leftrightarrow \quad \exists s'.\, [s' \in T(R)(s) \;\wedge\; s' \| \models q(\bar{\text{Y}})]$$

Proof

Part a: $s \| \models wp(R,\, q(\bar{\text{Y}})) \quad \rightarrow \quad T(R)(s) \neq \emptyset \wedge \; \forall s'.[s' \in T(R)(s) \rightarrow s' \| \models q(\bar{\text{Y}})]$

Proposition 4.22 says that$\forall$ fixed $\bar{\text{Z}}$ free in q:

$$s \| \models wp(R,\, q(\bar{\text{Z}})) \quad \rightarrow \quad \exists s'.s' \in T(R)(s)$$

We now show that $\forall$ fixed $\bar{\text{Z}}$ free in q:

$$s \| \models wp(R,\, q(\bar{\text{Z}})) \quad \rightarrow \quad \forall s' \in T(R)(s).).r\; s' \| \models q(\bar{\text{Z}})$$

Let us call G $g2f$(GUARD), B the BODY, T the TELL, and C the CONSUME.

$s' \| \models \neg\, q(\bar{\text{Z}}) \;\; \wedge \;\; s' \in T(R)(s) \;\rightarrow$

$s \| \models ep(R, \neg\, q(\bar{\text{Z}})) \;\leftrightarrow$

$s \| \models \exists \bar{\text{X}}.(\text{G} \wedge wp(\text{C}, wp(\text{B}, \{wp(\text{T}_1, \neg\, q(\bar{\text{Z}})),\, wp(\text{T}_2, \neg\, q(\bar{\text{Z}}))\}))) \;\leftrightarrow$

$s \| \models \exists \bar{\text{X}}.(\text{G} \wedge wp(\text{C}, wp(\text{B}, \{\neg\, wp(\text{T}_1, q(\bar{\text{Z}})),\, \neg\, wp(\text{T}_2, q(\bar{\text{Z}}))\}))) \;\leftrightarrow \qquad$ (let $q_i \equiv wp(\text{T}_i, q(\bar{\text{Z}}))$)

$s \| \models \exists \bar{\text{X}}.(\text{G} \wedge wp(\text{C}, wp(\text{B}, \{\neg q_1, \neg q_2\}))) \;\leftrightarrow$

$s \| \models \exists \bar{\text{X}}.(\text{G} \wedge wp(\text{C}, \exists \bar{\text{Y}}.[\text{B}(\bar{\text{X}}, \bar{\text{Y}}) \wedge \neg q_1] \vee [\nexists \bar{\text{Y}}.\text{B}(\bar{\text{X}}, \bar{\text{Y}}) \wedge \neg q_2]))$

On the other side:

$s \| \models wp(R, q(\bar{\text{Z}})) \;\leftrightarrow$

$s \| \models \exists \bar{\text{X}}.\text{G} \;\; \wedge \forall \bar{\text{X}}.[\text{G} \rightarrow wp(\text{C}, wp(\text{B}, \{q_1, q_2\})))] \;\leftrightarrow$

$s \| \models \exists \bar{\text{X}}.\text{G} \;\; \wedge \forall \bar{\text{X}}.\, [\text{G} \rightarrow wp(\text{C}, \exists \bar{\text{Y}}.[\text{B}(\bar{\text{X}}, \bar{\text{Y}}) \wedge q_1] \vee [\nexists \bar{\text{Y}}.\text{B}(\bar{\text{X}}, \bar{\text{Y}}) \wedge q_2])] \;\leftrightarrow$

$s \| \models \nexists \bar{\text{X}}.[wp(\text{C},\, \nexists \bar{\text{Y}}.[\text{B}(\bar{\text{X}}, \bar{\text{Y}}) \wedge q_1] \wedge [\exists \bar{\text{Y}}.\text{B}(\bar{\text{X}}, \bar{\text{Y}}) \vee q_2])]$

Leading to a contradiction.

Part b: $s \| \models ep(R,\, q(\bar{\text{Y}})) \quad \rightarrow \quad \exists s'.\, [s' \in T(R)(s) \;\wedge\; s' \| \models q(\bar{\text{Y}})]$

Let us consider the case of a rule without the body, the proof for the general case is an extension exploiting the same ideas. $\forall$ fixed $\bar{\text{Y}}$ free in q:

$$s \| \models ep(R,\, q) \quad \overset{def\_4.13}{\equiv} \quad s \| \models \exists \bar{\text{X}}.\, [\text{G} \;\wedge\; C(q)\,]$$

where $C(q) = wp(\text{CONSUME}, wp(\text{TELL}, q))$, i.e. a syntactic transformation of q, $\text{G} = g2f(\text{GUARD})$, and $\bar{\text{X}}$ is the set of the variables free in G.

By definition, all the variables appearing in the TELL and in the CONSUME of a rule also appear in the rule GUARD. Hence, by definition 4.7 (pag. 21), $\bar{\text{X}}$ covers all the variables free in $C(q)$.

Moreover, (see definition 4.5, pag. 21), $\bar{\text{X}}$ are all the variables appearing in the rule guard in positive conditions.

$s\|\!\!= \exists\bar{\text{X}}.\,[\,\text{G} \ \wedge \ C(q)\,] \quad \equiv$ for ϑ grounding for $\bar{\text{X}}$

$s\|\!\!= \exists\vartheta.\ (\text{G}\vartheta \ \wedge \ C(q)\vartheta\,) \quad \equiv$ (definition 3.8, pag. 13)

$\exists\vartheta.\ s\|\!\!= (\text{G}\vartheta \ \wedge \ C(q)\vartheta)$

Let us consider now the rule instance $R\vartheta$. We have, by definition 4.5 (pag. 21): $\text{G}\vartheta = g2f(\text{GUARD}\vartheta) = g2f(\text{GUARD})\vartheta = \text{G}\vartheta$. Moreover there are no free variables in $\text{G}\vartheta$, and there are no free variables in $C(q)\vartheta$. Hence, by definition 4.7 (pag. 21) and definition 4.13 (pag. 25) ($R\vartheta$ is actually deterministic):

$$s\|\!\!= \text{G}\vartheta \ \wedge \ C(q)\vartheta \quad \equiv \quad s\|\!\!= ep(R\vartheta,\, q) \quad \equiv \quad s\|\!\!= wp(R\vartheta,\, q) \tag{1}$$

Now:

$s\|\!\!= wp(R\vartheta,\, q) \quad \rightarrow$ see part a

$T(R\vartheta)(s) \neq \emptyset \wedge \ \forall s'.[s' \in T(R\vartheta)(s) \rightarrow s'\|\!\!= q] \quad \rightarrow$

$\exists s'.\,[s' \in T(R\vartheta)(s) \wedge s'\|\!\!= q]$

We conclude showing that $T(R\vartheta)(s) \subseteq T(R)(s)$

Let $\bar{s} \in T(R\vartheta)(s)$, by definition 4.17 (pag. 27):

$\forall p.\,\forall\bar{\text{Z}}$ free in $p.\ \bar{s}\|\!\!= p \ \rightarrow \ s\|\!\!= ep(R\vartheta,\, p)$

Now:

$s\|\!\!= ep(R\vartheta,\, p) \quad \rightarrow$ see (1)

$s\|\!\!= \text{G}\vartheta \wedge C(p)\vartheta \quad \rightarrow$

$s\|\!\!= \exists\vartheta\,(\text{G}\vartheta \wedge C(p)\vartheta) \quad \rightarrow$ (definition 4.13, pag. 25, ϑ is grounding for G and $C(p)$)

$s\|\!\!= ep(R,\, p)$

Hence $\forall p.\,\forall\bar{\text{Z}}$ free in $p.\ \bar{s}\|\!\!= p \ \rightarrow \ s\|\!\!= ep(R,\, p)$, and so $\bar{s} \in T(R)(s)$

Part c: $s\|\!\!= ep(R,\, q(\bar{\text{Y}})) \quad \leftarrow \quad \exists s'.\,[s' \in T(R)(s) \ \wedge \ s'\|\!\!= q(\bar{\text{Y}})]$

Derives directly from definition 4.17 (pag. 27).

Part d: $s\|\!\!= wp(R,\, q(\bar{\text{Y}})) \quad \leftarrow \quad T(R)(s) \neq \emptyset \wedge \ \forall s'.[s' \in T(R)(s) \rightarrow s'\|\!\!= q(\bar{\text{Y}})]$

Derives from part c and Proposition 4.21 (pag. 27). □