Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

An Abstract Interpretation Framework for Diagnosis and Verification of Timed Concurrent Constraint Languages

Candidate:
Laura Titolo

Supervisors:
Prof. Marco Comini
Prof. Alicia Villanueva García

May 2014

Author's e-mail: laura.titolo@uniud.it

Author's address:

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

To all the people I love.

Abstract

Nowadays, concurrent and reactive systems are widely used in a great variety of modern applications. We can think, for example, in mobile phone applications or in the software used in the medical or in the financial sphere. It is often the case that these applications are classified as critical, therefore a single error in the system can lead to catastrophic consequences. It is well known that finding program bugs is a very hard task, which become even worse when it is necessary to deal with time and concurrency features. For these reasons, formal verification of concurrent and reactive systems is a hot topic in modern computer science.

The concurrent constraint paradigm (*ccp* in short) is a simple but powerful model for concurrent systems. It is different from other programming paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model. In this way, the languages from this paradigm can easily deal with partial information (an underlying constraint system handles constraints on system variables).

Within the *ccp* family, the *Timed Concurrent Constraint Language* (*tccp* in short) adds to the original *ccp* model the notion of time and the ability to capture the absence of information. With these features, it is possible to specify—in a very natural way—behaviors typical of reactive systems such as *timeouts* or *preemption* actions.

The existing formal techniques for the verification of *tccp* are based on model checking. Model checking is a verification method that, given a graph representation of the program and a temporal logic formula, is able to check if the program satisfies the formula. However, this method suffers the state-explosion problem, i.e., the dimension of the graph grows exponentially w.r.t. the dimension of the program. This problem limits the use of model checking, especially in presence of concurrency.

In the field of formal verification, abstract interpretation is a valid alternative to model checking. Abstract interpretation is a theory of sound semantic approximation proposed with the aim of providing a general framework for analysis, verification and debugging of systems. The main idea behind this approach is to approximate the program behavior (or concrete semantics) into an abstract semantics in order to obtain effectiveness and efficiency at the price of losing some precision in the results.

In this thesis, we propose a semantic framework for *tccp* based on abstract interpretation with the main purpose of formally verifying and debugging *tccp* programs.

A key point for the efficacy of the resulting methodologies is the adequacy of the concrete semantics. Thus, in this thesis, much effort has been devoted to the development of a suitable small-step denotational semantics for the *tccp* language to start with.

Our denotational semantics models precisely the small-step behavior of *tccp* and is suitable to be used within the abstract interpretation framework. Namely, it is defined in a compositional and bottom-up way, it is as condensed as possible (it does not contain

redundant elements), and it is goal-independent (its calculus does not depend on the semantic evaluation of a specific initial agent).

Another contribution of this thesis is the definition (by abstraction of our small-step denotational semantics) of a big-step denotational semantics that abstracts away from the information about the evolution of the state and keeps only the the first and the last (if it exists) state. We show that this big-step semantics is essentially equivalent to the input-output semantics defined by de Boer, Gabbrielli and Meo in [43].

In order to fulfill our goal of formally validate *tccp* programs, we build different approximations of our small-step denotational semantics by using standard abstract interpretation techniques. In this way we obtain debugging and verification tools which are correct by construction. More specifically, we propose two abstract semantics that are used to formally debug *tccp* programs. The first one approximates the information content of *tccp* behavioral traces, while the second one approximates our small-step semantics with temporal logic formulas. By applying abstract diagnosis with these abstract semantics we obtain two fully-automatic verification methods for *tccp*.

Acknowledgments

Finishing a PhD is a milestone for someone who strives for an academic career, but, in my case, I'd rather like to consider it as the conclusion of an intense chapter of my life and a prelude to another one full of new adventures and feelings. For this reason, I would like to thank all the people that have had a special role for me in these last years and have helped me in so many different ways to reach this important goal.

First of all, I express my gratitude to my advisors, Marco Comini and Alicia Villanueva: they have made a great team to me. Marco started advising me from the very beginning of my university career and I owe him a great part of my knowledge in computer science (including all the secrets for defining good macros in LaTeX). Alicia's guidance was also essential, she was very supportive and willing to help me even with the most practical details of my thesis.

I'm also grateful to María Alpuente and all the ELP group to welcome me as their colleague and gave me the opportunity to travel and assist to schools and conferences around the world.

Thanks also to Carlos Olarte, Moreno Falaschi, Paqui Lucio, Jose Gaintzarain, Agostino Dovier, Vijay Saraswat and Frank Valencia for some interesting discussions about my thesis and their useful advices.

However, I would not have reached this important goal without the full support and love of my parents, Lorella and Luigi. They have always gave me everything I needed to fulfill my dreams and follow my passions and they are the "responsible" for raising me as the person I am right now.

Thanks to all my family: uncles, aunts and cousins (I have a lot!), but especially to my grandmother Natalina for having raised me with all her love and sweetness and also for having pampered me a little bit.

A huge "thank you" goes to my *amore* Marco, for standing by my side and loving me all these years, for putting up with me even in my worst rompi-mode, for sharing with me beautiful moments (and his closet), for our funny and tiring trips (like the tour-de-force week-end in Paris, my unlucky journeys in Granada or our adventure in Sardinia with the Mirto's bottle and the horse), for his company and support in a lot of first aid queues, for scaring me when he mimics Twin Peaks' Bob, for our Breaking Bad/Game of Thrones marathons and, above all, for letting me become part of his life and for believing that we can better our love every day.

I want to thank my best friends Valeria and Silvia for our enduring friendship which didn't change despite the distances and our different obligations in life, for growing up (or better old) together and being just us at the same time, for a lot of great trips and holidays in Bibione and Premantura (risking our lives due to our questionable driving skills), for having come to visit me in Valencia during my Erasmus bringing with them the Maniago

rain cloud, for giving me the most beautiful birthday gift in the world: the Roberto Bolle's book, and for being always here when I need them.

Thanks to my "wife" Marinella for our endless Gossip Girl/The Vampire Diaries night sessions, for sharing with me the gym fixation, for our healthy and elaborate dinners at home, for our laugh before going to bed, and, above all, for being like a sister to me in these last two years in the Messina flat.

Thanks also to Itziar for being my very first friend in Valencia, for sharing with me many passions and (very intellectual) hobbies (such as make-up and fashion jewelry) and also for our unforgettable bellydance adventure in Bologna.

These PhD years would not have been so funny without the amazing Vi.Ci.Na.Ti. team! We had a lot of fun (and spritz and americani) together and they have made worth going to work everyday.

Francesca Nadalin was my office mate in these three years, we shared a lot of new experiences together and we became very close friends. I thank her for having been so nice and supportive to me, for all the fun we had in and outside our office and for our great trip to Berlin visiting Alex (thank you also for your delicious handmade dessert, Fra!).

If some years ago someone had told me that I would have become friend with Matteo "Cicu" Cicuttin I would have never believed it. But luckily now we are friends and I want to thank him for helping and supporting me in a lot of difficult times and for his contagious passion and unceasing curiosity for science and research.

Thanks also to Riccardo "Vice" Vicedomini for his kindness and hilarity and for being the guy always present (and above all punctual) at our aperitivi.

I want to thank also Emanuela Pitassi, my other office mate, for our conversations about life, about our PhD troubles and gossip almost every friday evening at tea time.

A special thanks goes to Alex Tomescu, he was my first office mate and we quickly became friends, he was very helpful and kind to me in my first PhD year and we enjoyed together a lot of cultural and fine arts activities (did you really enjoy Pina Bausch film, Alex?).

Thanks also to Luca Torella who have shared with me the thesis advisor, his time in Valencia and also the great and funny experience of the Spring School in Bertinoro (do you remember the "fake professor" and the Kowalski catnap in the bus?).

I want to thank also other people at the University of Udine: Giovanni and Giorgio Bacci for the unavoidable mensa lunches and the endless discussion about investigative reports, Paolo Baiti and Andrea Baruzzo for your company at lunch, Giulia Giordano for your conversations, Riccardo Sioni and Andrea Vianello for your kindness and for converting me in a guinea pig for your experiments.

I also want to thank many people that I met at the University of Valencia which made my stays so pleasant: Sonia Santiago for her kindness and help, Raúl Gutiérrez for his company and goodness, Germán Vidal for his bizarre conversations and his photography tips, David Insa for being such a good office mate to me and also to Santiago Escobar, María José Ramirez, Salvador Lucas, Josep Silva, Daniel Romero, Alexei Lescaylle, Javier Insa, Javier Espert, Francisco Frechina, Demis Ballis, Fernando Martínez, Julia Sapiña, Salvador "Tama" Tamarit, Pepe Iborra and Fernando Tarín.

Thanks also to Andrea "Tello" Tellini for being such a good friend, for watching together (or at least commenting in real-time via WhatsApp) figure skating competitions, for the fun we had at the Spanish lessons with Marilú, for his hospitality in Madrid, and for being an example of hard work and study from our first day at SUPE. Thanks also to my

other SUPE friends: Nunzio, Emanuele, Giacomo, Gabriele, Michele, Caterina, Simone, Stefano, Sara, Martina...

I want to thank also my Uniud friends: Lisa for her sweetness and kindness, Ambra for her company and amusing conversations about everything, Caterina for her passion and congeniality, and finally Marco P. for being capable to make me smile even in the worst moments and for having always been very supportive to me.

Thanks to Raffy, Elisa and Erika for being my oldies but goodies friends in Maniago, I still remember with affection how we enjoied skating and bicycling in Area Violis in those distant summer evenings.

I want to thank Alex and Simona for sharing with me and Marco a lot of good moments in Valencia, they are like family to us. Thanks also to Marco's mum, Marta, for her help and care.

Thanks to Annuccia for her energy and joy, for being my youngest flat mate and for trying to take me and Marinella out to party even when we were really busy learning English with our TV series, and also thanks to Gianluigi for his kindness even if we spent very little time together in the Messina flat.

Thanks to my dance teachers, Simona and Teresa, for transmitting me their passion, and for their lessons that have been a happy escape from all the stress caused by my thesis and my research. Thanks to all my dance mates too (both in Udine and Valencia) for a lot of great moments and fun we had together.

I want to thank my first dance teacher Rita Gentile for teaching me to be more disciplined and self-confident, and to help me start believing in myself.

Thanks also to my first computer science teacher Marco Giacomini to help me understand what computer science really is and to suggest me to follow this path.

I would like to thank all my friends in Italy: Matteo, Luca DB, Vera, Alberto, Centa, Birin, Matteo G., Nicolino and Ilaria, Lorenzo the Doctor, Riccardo, Tommaso, Ricky, Davide, Mattia, Willy, Lara, Matteo P. and all the others that I've forgotten.

Thanks also to all my friends in Spain: Pilar, Barbara, Luis, Mar, Abel, George, Isabel and their baby Raúl, Alicia, Raúl, Alba and Julia (to appear), Raul MJ, Héctor, Nekro, Guti, Rúben, Rata and Alejandra, Xisco, Clara, Mireia and Luis, and more.

I am grateful to each and every one of you with all my heart, everyone of you helped me in some way or another, consciously or not, to make this chapter of my life unique, thank you!

May 2014,

Laura Titolo

Contents

List of Figures

Introduction

"Contradiction is not a sign of falsity, nor the lack of contradiction a sign of truth."

Blaise Pascal

In the last years, concurrent, reactive and distributed systems have had a wide spread. Time aspects have become essential to an increasingly large number of applications and the large diffusion of internet have made fundamental the use of concurrency and distributed features in modern software.

A *concurrent system* contains different components that run in parallel and interact with each other. When these components are located in different parts of the world, the system is called *distributed*. *Reactive systems* are those systems that interact continuously with their environment and that require the specification of some timing constraints, for example, that a certain signal is expected in a bounded period of time. Among this family we can distinguish *real-time systems* (e.g. process controllers, signal processing systems) that are subject to hard timing constraints. A reactive system can be seen as a concurrent one where the main system and the environment are two agents that run in parallel and exchange information.

Often, these systems are classified as critical, i.e., a single error in the software can lead to great loss in human lives or money. We can think, for example, on electronic financial transitions, electronic commerce, medical instruments, or air traffic control. In [22, 79], some significant examples of error cases are listed. When the software is critical, it is necessary to use formal methods to debug and verify it, in order to be sure that the system behaves correctly. Also in case the software is not critical is recommended to use formal methods in order to detect bugs. In fact, concurrency bugs are among the most difficult to find since they result from the concurrent contribution of several agents that run in parallel. Furthermore, interleaving and scheduling features additionally complicate the debugging phase since they generate computations which are hard to reproduce.

Formal methods are a collection of notations and techniques for describing, verifying and analyzing systems. Applying formal techniques usually requires modeling the system first. A model of a system is a mathematical representation of the properties that are of interest, by keeping the essential details and by omitting unimportant aspects.

Many formalisms have been developed to model concurrent systems. Some examples of synchronous models are the process calculi of Milner's CCS [82], Hoare's CSP [70] or the ACP of Bergstra and Klop [9]. One of those formalisms is the *Concurrent Constraint* (*cc*) *paradigm* [104]. It differs from other paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model. Thanks to this notion, it is possible to manage easily partial information since an underlying constraint system handles constraints on system variables.

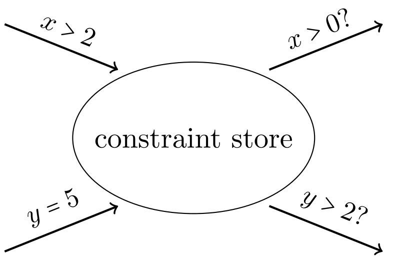


Figure 1: Example of actions on a constraint store in *ccp*

I.1 Modeling concurrent and reactive systems within the *cc* paradigm

Concurrent systems can be seen as a set multiple computing *agents* (also called *processes*) that interact among each other. Some examples of these systems are communication systems based on *message-passing*, communication systems based on *shared-variables*, or *synchronous* systems.

There are many models for concurrent systems. The model considered in this thesis is the *Concurrent Constraint paradigm* (*cc* paradigm or *ccp*) defined by Saraswat and Rinard in [103, 104, 109] as a simple and powerful model of concurrent computation. The *cc* paradigm is parametric w.r.t. a *constraint system*. A constraint system can be seen as a partial information system (see [110]): instead of knowing the specific value of a variable, just partial information is available. Thus, in this computational model, the notion of *store-as-valuation* from von Neumann is replaced with the notion of *store-as-constraint*. In this formalism, the agents exchange information through a global constraint store that is common to all agents. Agents can add new information in the global store, and query about its content.

A few years after the introduction of *ccp*, Saraswat, Jagadeesan and Gupta defined an extension over time of the *cc* paradigm. This new language, called *Temporal Concurrent Constraint* (*tcc*) language ([105, 106]) was inspired by synchronous languages such as ESTEREL [11], Lustre [18] or SIGNAL [61]. *tcc* is able to specify *reactive systems*, especially real-time and embedded ones (a small device designed for specific control functions within a larger system). The key idea was to introduce a notion of *discrete* time and some constructs which allow to model notions such as *time-outs* or *preemptions*. A time-out waits for a limited period of time for an event to occur, if this event does not happen, then an exception is executed. A preemption consists in the ability of detecting an event and, as a consequence, aborting the current process and executing a new one. As pointed out in [106], the essence of the time-out and preemptions mechanisms is in the ability to detect the *absence* of an event, as well as its presence.

Another extension over time was presented in [108]: the *Timed Default Concurrent Constraint programming*. This language allows one to model *strong preemptions*: the abort of the current process and the execution of the new one must happen at the same time of the detection of the event. As pointed out in [10] and [108], there are some critical applications in which this kind of preemption is required.

In 1998 Gupta, Jagadeesan and Saraswat presented a language which incorporates a notion of *continuous* (or *dense*) time into the *cc* model: the *Hybrid cc* (*hcc*) language [65]. The *hcc* language is able to model *hybrid systems* which are systems that have a continuous behavior controlled by a discrete component. For example, a thermostat can be seen as a hybrid system: a continuous variable models the temperature, and the turn-on or turn-off of the system depends on the temperature value limits. [66] shows some applicative examples for the Timed Default Concurrent Constraint programming language and for the *hcc* language.

In 1999, de Boer, Gabbrielli and Meo presented a different approach to extend the *cc* paradigm with a notion of discrete time inspired by the process algebra model. The *Timed Concurrent Constraint programming* (in short *tccp*) [43] adds to *ccp* the notion of time and some constructs that check for the absence of information in the constraint store, allowing one to implement behaviors typical of reactive systems, as in *tcc*.

Although *tcc* and *tccp* are both extensions of *ccp*, the first one is inspired by the synchronous languages approach, while the second one is inspired by process algebras. Therefore, these two languages have some important differences.

First of all, the two languages differ on the notion of time. In *tcc* the computation proceeds in bursts of activity and in each phase a deterministic *ccp* process is executed to respond to an input produced by the environment. This process accumulates monotonically information in the constraint store until it reaches a resting point, i.e., a terminal state in which no more information can be generated. When this resting point is reached, the current process can trigger the actions in the next time phase. It follows that each time interval is identified with the time needed for a *ccp* process to terminate its computation. On the contrary, in *tccp* a global discrete clock is introduced. A single time unit corresponds to the time that a process takes to perform a constraint store elementary action (adding information or querying the global constraint store).

Another difference regards the recursion and the Turing completeness of these languages. On one hand, *tcc* allows only procedures without parameters. As explained in [87], this kind of recursion is equivalent to replication in terms of expressive power, thus, *tcc* is not a Turing powerful language. On the other hand, *tccp* is Turing powerful since it allows recursion with parameters.

The interpretation of the parallel operator is different in the two languages: *tcc* interprets the parallelism in terms of interleaving, while *tccp* makes the assumption of infinite processors and uses the notion of maximal parallelism.

Finally, *tcc* is a deterministic language, while *tccp* allows non-determinism. The notion of non-determinism was introduced into the *tcc* model a few years later by Palamidessi and Valencia in [93] by defining the *ntcc* language.

In 2007, Olarte, Palamidessi and Valencia introduced the Universal Timed Concurrent Constraint language (*utcc*) [89, 90] with the aim of modeling mobile reactive systems. *utcc* extends *tcc* with the notion of mobility in the sense of Milners π-calculus [83, 84]. In this way this language allows the generation and communication of private channels or links.

The extensions of the *cc* paradigm proposed in the literature are graphically illustrated in Figure 2.

$$\begin{tikzcd}
 & & ccp \arrow[dl] \arrow[d] \arrow[dr] & \\
 & tcc \arrow[dl] \arrow[d] \arrow[dr] & tccp & hcc \\
\text{default } tcc & ntcc & utcc &
\end{tikzcd}$$

Figure 2: The languages of the *cc* paradigm

I.2 Formal Verification

The verification of a system consists in checking its correctness with respect to a given intended behavior. In this section, we present some of the principal techniques that have been proposed in the literature. We distinguish between non-formal techniques (such as testing), which cannot assure the absence of errors, and formal techniques, that are based on some mathematical theory and can assure that a program behaves as expected.

Testing [75, 86, 95] is a simple technique to check the correctness of a program. It consists in executing the program that we want to verify, and then analyzing the executions to detect errors. Each execution is compared with the expected one and, in case they do not coincide, it means that an error has occurred. This technique is widely used to improve the quality of software, since it can be used by non expert people in mathematics or logic. Testing is not considered a formal verification method since it is based on the analysis of only some executions of the considered program. Therefore, it is not possible to ensure the total absence of errors.

Formal verification is a set of mathematical notations and techniques whose aim is that of proving that a program satisfies a given specification (and thus does not contain specific errors).

Theorem proving was the first formal verification method studied in the literature (see [56, 69]). This is a deductive method which is guided by the user and, originally, had to be performed manually. Therefore, the success of deductive proofs depends a lot on the capability of the user, since the verification process can be difficult and error prone. For this reason, classic theorem proving must be applied by expert people. In order to solve these problems, some tools have been developed to make this process semi-automatic. Some of the most used automated theorem provers (or proof assistant) are Isabelle [96], Coq [30] and PSV [92]. These tools, by using some heuristics, are able to suggest the user how to continue a proof at a specific point. Automated theorem proving has a lot of good features: it is reliable since it uses mathematics and logic theory, it admits the introduction of invariants in the code to allow run-time verification, and it can help one to define formal semantics of programming languages. However, this approach has also many drawbacks, for example it is not completely automatic and a lot of time and effort are needed to complete a proof.

One of the most popular verification methods is *model checking*, introduced by Clarke and Emerson [20, 21, 48] and by Queille and Sifakis [100] independently. Model checking is an automatic technique that, given a graph representation of the program and a temporal logic formula, checks if the program satisfies the formula by performing an exhaustive analysis of the system state-space. However, this method suffers the *state-explosion problem*, i.e., the dimension of the graph grows exponentially w.r.t. the dimension of the program. Thus, classic model checking is ineffective in most cases. Some strategies have been presented in the literature to mitigate this problem. *Abstract model checking* uses an approximation of the model that removes some irrelevant details in order to reduce its size, while *symbolic model checking* [17, 81] uses an implicit representation of the model based on Ordered Binary Decision Diagrams (OBDDs), defined in [16]. The key idea is that the temporal formula can be checked directly over the implicit representation of the system in a more efficient way. Another drawback of model checking is that, in general, the system must be manually modeled in the language handled by the model checker, and this is sometimes a difficult and error prone task. In the framework of formal methods, an alternative approach to model checking can be found in abstract interpretation theory.

Abstract interpretation [31, 33] is a general theory for approximating the behavior of a program. The relevant feature of abstract interpretation is that, once the property intended to be observed has been modeled by an abstract domain, we have a methodology to systematically derive an abstract semantics, which allows us to effectively compute a (correct) approximation of the behavior of the program. This is obtained in general at the expense of precision.

Another verification technique is *declarative debugging* that was firstly proposed for logic programs [54, 76, 113] and then extended to other languages. It is a semi-automatic debugging technique where the debugger tries to locate the node in a computation tree which is ultimately responsible for a visible bug symptom. This is done by asking questions on correctness of solutions to the user, which assumes the role of the oracle.

Abstract diagnosis [25] is an instance of declarative debugging based on abstract interpretation. Given a program and a finite approximation of the intended behavior, this method *automatically* checks if the program behaves correctly. Abstract diagnosis uses an abstract semantics, defined as the fixpoint of an (abstract) immediate consequence operator, and a given approximated intended behavior, called *abstract specification*. Errors are detected by comparing the abstract specification with the result of the calculus of just one step of the abstract immediate consequence operator by assuming the abstract specification to be correct.

I.2.1 Formal verification of timed concurrent constraint languages

In this section we provide an overview of the state of the art related to formal verification of the time extensions of the *cc* paradigm.

The existing formal techniques for the verification of *tccp* are based on model checking [53]. As already pointed out, the main drawback of this technique is the combinatorial blow up of the state-space. This problem, called *state-explosion problem*, becomes even worse for concurrent systems. To mitigate this problem two different approaches were proposed: abstract model checking in [5, 4] that reduces the size of the initial model by means of an approximation, and symbolic model checking in [2] that uses a symbolic representation of the model. Although these methods enhance the applicability of *tccp* model checking, the combinatory explosion of the state-space is still a problem.

In [51], a first approach to the declarative debugging of a timed *ccp* language is presented. Falaschi *et al.* introduce a semantic framework for *ntcc* and, by using standard abstract interpretation techniques, they define an automatic debugging method. In 2009 a similar approach was presented in [50] for *utcc*.

In [44, 45], a temporal logic is introduced for reasoning about *tccp* programs, joint to a sound and complete proof system. This logic is an extension of the Linear Temporal Logic (LTL) presented by Manna and Pnueli in [78]. The authors replace classical logic propositions with constraints and add the notion of monadic modalities over constraints. Monadic modalities take the form $X(c)$, where c is a constraint of the underlying constraint system, and X expresses a specific property of the program. This extension is needed by the authors of [45] to distinguish the stimuli coming from the environment and the information produced by the program itself in response to the environment.

An analogous work was made for *ntcc*. In [93, 88] *ntcc* is equipped with a temporal logic, called CLTL (Constraint LTL), able to express program properties. The authors also provide a satisfiability relation for the formulas w.r.t. the behavioral sequences and a proof system to check the properties of *ntcc* processes. CLTL slightly differs from the logic presented in [45] since it does not use monadic modalities. In this context, these modalities are unnecessary because the authors are interested in reasoning about the strongest postcondition, thus they abstract away from the inputs of the external environment.

In [116], Valencia presents some decidability results for the verification of *ntcc* programs using CLTL specifications. He shows that for the locally-independent fragment of *ntcc*, it is possible to automatically verify a negation-free CLTL formula. Namely, if the semantics of a given program P is equivalent to the one of a program P_ϕ, which is built from the formula ϕ by means of a correct procedure, then P satisfies ϕ. The semantic equivalence for the locally-independent fragment of *ntcc* is decidable since there is only a finite number of possible configurations of the constraint store for a locally-independent *ntcc* program. This is a consequence of the monotonicity of the locally-independent fragment of the language and the absence of recursion (only the replication operator is admitted).

I.3 Automatic approaches based on Abstract Interpretation

Abstract interpretation [31, 33] is a general theory for approximating the semantics of discrete dynamic systems, which was originally developed by Patrick and Radhia Cousot in the late 70's as a unifying framework for specifying and validating static program analyses. In this theory, the behavior (or concrete semantics) of the system is approximated by means of an *abstract semantics* which models only the interesting properties on the program execution. An abstract semantics is built by replacing operations in a suitable concrete semantics with the corresponding abstract operations defined on data descriptions, namely, *abstract domains*. Such domains are called abstract because they abstract, from the concrete computation domain, the properties of interest.

The relevant feature of abstract interpretation is that, once the property intended to be observed has been modeled by a suitable abstract domain, we have a methodology to systematically derive the correspondent abstract semantics, which in turn allows us to effectively compute a (correct) approximation of the property. By using this approach, most of the theorem-proving techniques, in the logical theory involved in program verification, boils down to computing on the abstract domain. This is obtained, in general, at the expense of precision.

The concrete and the abstract semantics are related by a pair of functions: the *ab-*

straction α and the *concretization* γ. The abstraction function approximates a concrete element into an abstract one, while the concretization function, given an abstract element a, returns the concrete elements which are approximated by a. This pair of functions is called Galois insertion.

The principal technical results of abstract interpretation theory are summarized in Section 1.3.

Abstract interpretation is inherently semantic sensitive, thus, different semantic definition styles lead to different approaches to program analysis and verification. The definition of an appropriate concrete semantics, capable of modeling the program properties of interest, is a key point in abstract interpretation [31]. This ability of a semantics to mimic exactly the program properties of interest is called *full-abstraction.* In the following section we will describe in more detail this concept and its related notions.

I.3.1 Behavioral equivalences, correctness and full-abstraction

The semantics of a program helps to understand its meaning and to reason about its behavior. A program admits different semantics depending on the computational properties we want to observe. This set of properties is called *observable behavior* and induces an *observational equivalence* on programs. Namely, two programs are equivalent w.r.t. a given property σ ($P_1 \approx_\sigma P_2$) if their behaviors cannot be distinguished on that property. For example, we can decide to observe how the state evolves at each step (small-step behavior), or what is the state of the computation at some specific points (big-step behavior) or even just the input-output behavior.

Semantics are useful to perform program analysis, verification and debugging. But, in order to apply semantic-based techniques, it is opportune to construct a formal denotational model that is able to capture the behavioral (or operational) properties of the program.

Given an equivalence on programs $\approx_\sigma$, it is possible to define a formal denotational semantics $\mathcal{S}^\sigma[\![P]\!]$ that models the operational behavior of the program w.r.t. the property σ. A semantics is said to be *fully abstract* w.r.t. a property σ when $P_1 \approx_\sigma P_2$ if and only if $\mathcal{S}^\sigma[\![P_1]\!] = \mathcal{S}^\sigma[\![P_2]\!]$. This means that it identifies all and only the programs which cannot be distinguished by σ. A non-fully abstract semantics includes non relevant aspects and introduces distinctions between programs that have the same behavior.

Notice that the notion of *full-abstraction* is different from the stronger notion of *equality* that is sometimes used in the literature. A semantics $\mathcal{S}$ is said to be *equal* to another semantics $\mathcal{S}'$ if $\mathcal{S}[\![P]\!] = \mathcal{S}'[\![P]\!]$ for any program P. Differently from the full-abstraction notion, the semantic equality requires the two compared semantics to be defined over the same denotation (or domain). It is easy to see that if two semantics are equal, then they are also fully-abstract, but not vice versa.

I.3.2 Semantic properties

Besides the full-abstraction, there are some other properties of the (concrete) semantics that are *particularly* relevant to obtain a denotational semantics suitable to be used for semantics-based analysis or verification and for having an effective and efficient implementation which computes an as-precise-as-possible abstract semantics.

Let us point out these properties and discuss about their benefits.

Goal-independent. A semantics has a *goal-independent* definition when the denotation of any compound (nested) expression is defined in terms of the denotations of most general calls. For instance, the semantics of an expression like $e \coloneqq f(g(v_1, v_2), v_3)$ is obtained by suitable semantics operators which, considering values v_1, v_2, v_3 and the semantics of $f(x, y)$ and $g(w, z)$, can reconstruct the proper semantics of e. Operational (top-down) semantics are rarely defined in this way since it is more natural (and easy) to give a (compositional) goal-dependent definition which produces the effects of the current expression that has to be evaluated. When one is interested in the results of the evaluation of a *specific* expression e, it would make little sense to define a more complicated goal-independent semantics formalization that first evaluates *all* most general expressions and then tailors such evaluations on e to mimic the effects of a top-down goal-dependent resolution mechanism. In the tailoring process, many parts of the computed denotations will not be used and thus much computation effort would be wasted.

However, when we are no longer focused on determining the actual evolution of a specific expression but we are interested in determining the properties of a program for all possible executions, things change radically. In this case, we necessarily have to determine the semantic information regarding all possible expressions, and then it is more economical to have a goal-independent definition and compute just the semantics of most general calls (and, when is needed, reconstruct from these the semantics of specific instancies).

Condensed. A semantics is *condensed* when denotations contain only the minimal necessary number of semantic elements that are needed to characterize the classes of semantically equivalent syntactic objects (or, in other words, the minimal information needed to distinguish a syntactic object x from the other syntactic objects that are not semantically equivalent to x).

This may not seem a useful property for a concrete semantics, which—in general—would nevertheless contain infinite elements even when is condensed. However, this reduction could anyway frequently change some infinite denotations into finite ones and—most important—*all* the abstractions of a condensed concrete semantics will inherit this property *by construction*. Hence, by having minimal (abstract) denotations, one obtains *by definition* algorithms that compute just the minimal number of (abstract) semantic elements. This is definitely a stunning advantage over non condensed approaches which rarely can regain this efficiency in some other way. One could argue that it would be possible to live with a simpler non-condensed concrete semantics and then, for each abstraction of interest, work on the specific case to find out its condensed representation. We find more economical (especially in the long run) to do the effort once for the concrete semantics and then obtain, by construction, that all abstractions are condensed (with no additional effort).

Bottom-up. A *bottom-up* definition (in addition to the previous properties), has also an immediate direct benefit for abstract computations. With a bottom-up definition, at each iteration we have to collect the contributions of all rules. For each rule we will use the join operation of the abstract domain in parallel onto all components of the body of the rule. With a top-down definition instead, we have to expand one component of the goal expression at a time, necessarily using several subsequent applications of the join operation (of the abstract domain) over all components, rather than a unique simultaneous join of all the semantics of components. The reduced use of the join of a bottom-up formulation has a twofold benefit. On one side, it speeds up convergence of the abstract fixpoint computation. On the other side, it considerably improves precision.

Compositional. *Compositionality* is one of the most desirable characteristics of a semantics, since it provides a foundation for *incremental* and *modular* verification. Compositionality depends on a (syntactic) program construction operator $\circ$, and holds when the semantics of the constructs $C_1 \circ C_2$ can be computed by composing the semantics of the components C_1 and C_2.

I.3.3 Abstract diagnosis and debugging

The time and effort spent on validation of computer programs is known to take over half of the total time for software production. Thus, debugging is an essential ingredient in software development.

The role of debugging, in general, is to identify and eliminate differences between the intended semantics of a program and its actual semantics. We will assume that the user has a clear idea about the results that should be computed by the program. An error occurs when the program computes something that the programmer did not intend (*incorrectness symptom*), or when it fails to compute something he was expecting (*incompleteness* or *insufficiency symptom*). In other words, incorrectness symptoms are answers which are in the actual program semantics but are not in the intended semantics, while incompleteness symptoms are answers which are in the intended semantics but are not in the actual program semantics.

Declarative debugging is a debugging technique which is concerned with model-theoretic properties. The idea behind declarative debugging is to collect information about what the program is intended to do and compare this with what it actually does. By reasoning from this, a diagnoser can find errors. The information needed can be found by asking the user a formal specification (which can be an extensive description of the intended program behavior or an older correct version of the program). The entity that provides the diagnoser with information is referred to as the *oracle*.

The declarative debugging method consists in two main techniques: incorrectness error diagnosis and insufficiency error diagnosis. The principal idea to find incorrectness errors is to inspect the *proof tree* constructed for an incorrectness symptom. To find the erroneous declaration the diagnoser traverses the proof tree. At each node it asks the oracle about the validity of the corresponding atom. With the aid of the answers the diagnoser can identify the erroneous declaration. Dually, insufficiency error diagnosis concerns the case when a program fails to compute some expected results. The objective for insufficiency diagnosis is to scrutinize the attempt to construct a proof for a result which incorrectly fails (or suspends).

An alternative approach to declarative debugging is *abstract diagnosis* which is a completely automatic debugging methodology based on abstract interpretation. It was originally proposed for logic programming [25] and later has been applied to other paradigms [1, 8, 51]. This approach allows one to reason only on the abstract properties of interest.

In this way, it simplifies the user the task of providing the specification. Abstract diagnosis can be considered as an extension of declarative debugging, since there are instances of the framework that deliver the same results. The intuition of the approach is that, given an abstract specification of the expected behavior of the program, one automatically detects the errors in the program. In order to achieve an effective method, abstract interpretation is used to approximate the semantics, thus results may be less precise than those obtained by using the concrete semantics.

Abstract diagnosis is parametric w.r.t. an abstract program property and it is based on the approximation of a concrete semantics expressed in terms of the least fixpoint of an immediate consequence operator $\mathcal{D}$.

The abstract diagnosis methodology can be described as follows. An "abstract immediate consequence operator" $\mathcal{D}^\alpha$ is obtained by approximating the concrete immediate consequence operator $\mathcal{D}$. Given the abstract intended specification $\mathcal{S}^\alpha$ of the behavior of the considered program P, we can check the correctness of P by a single application of $\mathcal{D}^\alpha[\![P]\!]$ and thus, by a static test, we can determine all the process declarations $d \in P$ which are wrong w.r.t. the considered abstract property.

Abstract diagnosis has some advantages w.r.t. declarative debugging. First of all, abstract diagnosis is a fully-automatic approach, while declarative diagnosis needs the intervention of an oracle during the debugging process. Abstract diagnosis avoids the need to provide symptoms in advance, while declarative debugging is a symptom driven approach. If an error symptom is caused by more than one bug, declarative debugging has to be reapplied in order to detect all the bugs related to the same error symptom. On the contrary, abstract diagnosis is able to detect all these errors in one single application of the method. The major drawback of abstract diagnosis is that, because of the abstraction, the method can lead to false positives. By approximating, we renounce to the precision of the obtained result, i.e., also correct parts of the code may be pointed out as erroneous. However, the main point is that erroneous code cannot be validated as correct, thus abstract diagnosis is suitable to validate critical systems in an efficient way. Using abstract properties as specifications is an advantage because it relieves the user from having to specify in excessive detail the program behavior, which could be more error prone than the coding itself. The choice of an abstract domain is, thus, a trade-off between the precision of errors that can be detected and the effort in providing the specification.

I.4 Thesis Approach

In this thesis, we propose a semantics based abstract interpretation framework for the language *tccp* in order to define debugging and verification tools for concurrent and reactive systems.

As said before, these systems strongly depend on time and interact continuously with the environment for an infinite period. For these reasons, the observation of the input-output behavior is not adequate since it is concerned only with finite computations and does not shows the evolution of the computation over time. We are interested, instead, in the small-step behavior, i.e., how the program evolves at each time instant, also for infinite computations.

As already pointed out, the definition of an appropriate concrete semantics, which models the behavior of interest, is a key point in abstract interpretation [31]. For this reason, in this thesis much effort has been devoted to the definition of a denotational semantics for *tccp* that is fully-abstract w.r.t. the small-step behavior of the language and that meets all the properties listed in Section I.3.2.

We propose a new compositional, condensed, goal-independent and bottom-up semantics for *tccp* that is fully-abstract w.r.t. the small-step behavior of the language and that is able to deal also with infinite computations. In our semantics, we associate, to each program a set of sequences representing its behavior in a condensed way. These sequences, called *conditional state traces*, contain the minimal information needed to represent a set of *tccp* behavioral traces, namely, the conditions that have to be satisfied at each time instant and the information introduced in the global store by the program. Due to its good properties, our semantics is shown to be suitable for verification and debugging purposes based on abstract interpretation.

We define also a big-step semantics (by abstraction of our small-step semantics) which tackles also outputs of infinite computations. We prove that its fragment for finite computations is (essentially) isomorphic to the traditional big-step semantics of [43]. Moreover, we also formally prove that it is not possible to have a correct input-output fixpoint semantics which is defined *solely* on the information provided by the input/output pairs (i.e., some more information into denotations is needed).

Another contribution of this thesis is the definition of a general abstract diagnosis scheme for *tccp* which is parametric to the desired properties to be verified. These properties are modeled by means of a suitable abstract domain which approximate the domain of conditional state traces.

We show two instances of this abstract diagnosis scheme by using two different abstract domains: a domain of abstract conditional state traces and a domain of temporal logic formulas.

In the first case, starting from our semantics, we deduce, using standard abstract interpretation techniques, an approximated semantics based on the abstraction of the underlying constraint system. The elements of the abstract domain are compact abstract traces which contain approximated information. This domain is suitable for verification features since it allows to express in a compact way the properties of both finite and infinite computations. Given a *tccp* program and an (abstract) behavior specification, we apply abstract diagnosis [25] to automatically detect if the program meets the desired specification. In this way, we obtain a fully-automatic verification method for *tccp*.

Similarly, we define an abstract semantics for *tccp* based on temporal logic formulas. In order to express *tccp* properties, we add constraints to the classical LTL by defining a new logic that we call csLTL (constraint system LTL). Intuitively, a csLTL formula represents the set of *tccp* conditional traces that satisfies that formula. As in the case of abstract traces, we can apply abstract diagnosis to obtain a fully-automatic verification method that checks if a *tccp* program satisfies a given formula. This method intuitively consists in viewing a *tccp* program P as a formula transformer and thus, in order to decide the validity of ϕ, we just have to check if the P-transformation of ϕ implies ϕ. The transformation has a cost which is linear in the program's size, and thus the computational cost of the whole method is due to the check of the implication.

In order to make the method effective we provide an automatic decision procedure for our csLTL logic. This procedure is the extension to csLTL of the tableau algorithm proposed in [58, 60] for propositional LTL.

I.5 Thesis Overview

The thesis is organized as follows:

In Chapter 1 we introduce the basic concepts, terminologies and notations used in this thesis.

In Chapter 2 the *tccp* language is presented in detail joint to its operational semantics which slightly differs from the original one defined in [43].

In Chapter 3 our new denotational semantics for *tccp* is introduced. We show its correctness and full-abstraction w.r.t. the operational small-step behavior. Illustrative examples for the main concepts are also included. In this chapter we also define the big-step semantics and we formally relate it to the original one of [43].

In Chapter 4 a general abstract diagnosis scheme for *tccp* based on Galois insertions is presented. By using standard abstract interpretation results we define a new abstract semantics for *tccp* which is correct by construction. This semantics associates a *tccp* program to a set of traces which contain approximated information. We exhibit an instance of the abstract diagnosis scheme that uses this abstract semantics and we show some examples of application of this method.

In Chapter 5 we present a different abstract diagnosis framework for *tccp* based only on the concretization function. This can be used in case the abstraction function cannot be defined. We instantiate this scheme with another abstract semantics which associates a *tccp* program to a csLTL formula and we show some examples of application of this method to find bugs in programs. We also present a tableau construction algorithm for csLTL to show the decidability of our proposal.

Finally, in Chapter 6 the implementation of our framework is discussed.

To improve the readability of the thesis, the most technical definitions, results and all the proofs can be found in the related chapter appendix.

I.6 Publications related to this thesis

In this section we list the publications related to this thesis.

In the work “Abstract Diagnosis for Timed Concurrent Constraint programs” [26] a first version of our small-step semantics for *tccp* (Section 3.1) is presented. Furthermore, a first approach to the abstract diagnosis of *tccp* programs (Chapter 4) is introduced. This work was presented at the 27th International Conference on Logic Programming (ICLP 2011) and then it appeared in the related special issue of the journal Theory and Practice of Logic Programming.

A first proposal of our tableau construction algorithm for csLTL (Section 5.5) was defined in the work “Towards an Effective Decision Procedure for LTL formulas with Constraints” [28] which was presented at the 23rd Workshop on Logic-based methods in Programming Environments (WLPE 2013).

The article “Abstract Diagnosis for *tccp* using a Linear Temporal Logic” [29] contains our abstract diagnosis approach based on temporal formulas together with the tableau construction algorithm that makes the method effective (Chapter 5). This article has been accepted for presentation at the 30th International Conference on Logic Programming (ICLP 2014) and it will appear in the correspondent special issue of the journal Theory and Practice of Logic Programming.

The article "A Condensed Goal-Independent Bottom-Up Fixpoint Modeling the Behavior of *tccp*" [27] contains the definition of our novel small-step semantics for *tccp* joint to the proof of full-abstraction w.r.t. the small-step behavior of *tccp* (Section 3.1). Furthermore, it contains the definition of our big-step semantics (Section 3.2). This work is currently under review for publication in the Journal Transaction on Computational Logic.

Finally, the article "An Abstract Interpretation Framework for Verification of Timed Concurrent Constraint Languages" [115] contains a summary of this thesis. It was presented at the Ninth ICLP Doctoral Consortium 2013 and then published in the online supplement of the journal Theory and Practice of Logic Programming.

1 Preliminaries

This chapter presents the basic notations and concepts we will use through this thesis. Some more specific notions will be introduced in the chapters where they are needed.

Sections 1.1 and 1.2 are taken from [24]. For the terminology not explicitly shown and for a complete introduction about fixpoint theory and algebraic notation, the reader can consult [15, 14, 77]. We will refer to [104] for further details on the concurrent constraint paradigm. In [78] the reader can find a complete introduction to Linear Temporal Logic.

1.1 Basic Set Theory

To define the basic notions we will use the standard (meta) logical notation denoting conjunction, disjunction, quantification and so on (*and*, *or*, *for each*, ...). For statements (or assertions) A and B, we will commonly use abbreviations like:

$\boldsymbol{A, B}$ for (A and B), the conjunction of A and B,

$\boldsymbol{A \Longrightarrow B}$ for (A implies B), or (if A then B), which express the logical implication,

$\boldsymbol{A \Longleftrightarrow B}$ for (A if and only if B), which expresses the logical equivalence of A and B.

We will also make statements by forming disjunctions (A or B), with the self-evident meaning, and negations (not A), sometimes written $\neg A$, which is true if and only if A is false.

A statement like $P(x, y)$, which involves variables x, y, is called a predicate and it becomes true when the pair x, y satisfies the property (or relation, or condition) modeled by P. We use logical quantifiers $\exists$ (read "there exists") and $\forall$ (read "for all") to write assertions like $\exists x. P(x)$ as abbreviating "for some x, $P(x)$" or "there exists x such that $P(x)$", and $\forall x. P(x)$ as abbreviating "for all x, $P(x)$" or "for any x, $P(x)$". The statement $\exists x, y, \dots, z. P(x, y, \dots, z)$ abbreviates $\exists x. \exists y. \cdots \exists z. P(x, y, \dots, z)$, and $\forall x, y, \dots, z. P(x, y, \dots, z)$ abbreviates $\forall x. \forall y. \cdots \forall z. P(x, y, \dots, z)$. In order to specify a set S over which a quantifier ranges, we write $\exists x \in S. P(x)$ instead of $\exists x. x \in S, P(x)$, and $\forall x \in S. P(x)$ instead of $\forall x. x \in S \Longrightarrow P(x)$.

1.1.1 Sets

Intuitively, a set is an (unordered) collection of objects. These objects are called elements (or members) of the set. We write $a \in S$ when a is an element of the set S. Moreover, we write $\{a, b, c, \dots\}$ for the set of elements $a, b, c, \dots$.

A set S is said to be a subset of a set S', written $S \subseteq S'$, if and only if every element of S is an element of S', i.e., $S \subseteq S' \iff \forall z \in S.\, z \in S'$. A set is determined only by its elements, so, sets S and S' are equal, written $S = S'$, if and only if every element of S is an element of S' and vice versa.

Sets and Properties

A set can be determined by a property P. We write $S := \{x \,|\, P(x)\}$, meaning that the set S has as elements precisely all those x for which $P(x)$ is true. We will not be formal about it, but we will avoid trouble like Russell's paradox (see [102]) and will have at the same time a world of sets rich enough to support most mathematics. This will be achieved by assuming that certain given sets exist right from the start and by using safe methods for constructing new sets.

We write $\varnothing$ for the null or empty set and $\mathbf{N}$ for the set of natural numbers $0, 1, 2, \ldots$. The cardinality of a set S is denoted by $|S|$. A set S is called *denumerable* if $|S| = |\mathbf{N}|$ and *countable* if $|S| \leq |\mathbf{N}|$.

Constructions on Sets

Let S be a set and $P(x)$ be a property. $\{x \in S \,|\, P(x)\}$ denotes the set $\{x \,|\, x \in S, P(x)\}$. Sometimes, we will use a further abbreviation. Let $E(x_1, \ldots, x_n)$ be an expression which represents a particular element for $x_1 \in S_1, \ldots, x_n \in S_n$ and $P(x_1, \ldots, x_n)$ is a property of such $x_1, \ldots, x_n$. We use $\{E(x_1, \ldots, x_n) \,|\, x_1 \in S_1, \ldots, x_n \in S_n,\, P(x_1, \ldots, x_n)\}$ to abbreviate $\{y \,|\, \exists x_1 \in S_1, \ldots, x_n \in S_n.\, y = E(x_1, \ldots, x_n),\, P(x_1, \ldots, x_n)\}$.

The *powerset* of a set S, $\{S' \mid S' \subseteq S\}$, is denoted by $\wp(S)$.

Let I be a set. By $\{x_i\}_{i \in I}$ (or $\{x_i \,|\, i \in I\}$) we denote the set of (unique) objects x_i, for any $i \in I$. The elements x_i are said to be *indexed* by the elements $i \in I$.

The *union* of two sets is $S \cup S' := \{a \,|\, a \in S \text{ or } a \in S'\}$. Let $\mathcal{S}$ be a set of sets, $\bigcup \mathcal{S} = \{a \,|\, \exists S \in \mathcal{S}.\, a \in S\}$. When $\mathcal{S} = \{S_i\}_{i \in I}$, for some indexing set I, we write $\bigcup \mathcal{S}$ as $\bigcup_{i \in I} S_i$. The *intersection* of two sets is $S \cap S' := \{a \,|\, a \in S, a \in S'\}$. Let $\mathcal{S}$ be a nonempty set of sets. Then $\bigcap \mathcal{S} := \{a \,|\, \forall S \in \mathcal{S}.\, a \in S\}$. When $\mathcal{S} = \{S_i\}_{i \in I}$ we write $\bigcap \mathcal{S}$ as $\bigcap_{i \in I} S_i$.

The *cartesian product* of S and S' is the set $S \times S' := \{(a, b) \,|\, a \in S, b \in S'\}$, the set of ordered pairs of elements with the first from S and the second from S'. More generally $S_1 \times S_2 \times \cdots \times S_n$ consists of the set of n-tuples $(x_1, \ldots, x_n)$ with $x_i \in S_i$ and S^n denotes the set of n-tuples of elements in S.

$S \setminus S'$ denotes the set where all the elements from S, which are also in S', have been removed, i.e., $S \setminus S' := \{x \,|\, x \in S, x \notin S'\}$.

1.1.2 Relations and Functions

A *binary relation* between S and S' ($R{:}\, S \times S'$) is an element of $\wp(S \times S')$. We write $x \mathrel{R} y$ for $(x, y) \in R$.

A *partial function* from S to S' is a relation $f \subseteq S \times S'$ for which $\forall x, y, y'.\, (x, y) \in f, (x, y') \in f \implies y = y'$. By $f{:}\, S \rightharpoonup S'$ we denote a partial function of the set S (the *domain*) into the set S' (the *range*). The set of all partial functions from S to S' is denoted by $[S \rightharpoonup S']$. Moreover, we use the notation $f(x) = y$ when there is a y such that $(x, y) \in f$ and we say $f(x)$ is defined, otherwise $f(x)$ is undefined. Sometimes, when $f(x)$ is undefined, we write $f(x) \in \aleph$, where $\aleph$ denotes the set of undefined elements. For each set S we assume that $\aleph \subseteq S$, $\aleph \cup S = S$ and $\varnothing \notin \aleph$. This will be formally motivated in Section 1.2.1.

Given a partial function $f\colon S \rightharpoonup S'$, the sets $supp(f) := \{x \in S \mid f(x) \text{ is defined}\}$ and $img(f) := \{f(x) \in S' \mid \exists x \in S.\, f(x) \text{ is defined}\}$ are, respectively, the *support* and the *image* of f. A partial function is said to be *finite-support* if $supp(f)$ is finite. Moreover, it is said to be *finite* if both $supp(f)$ and $img(f)$ are finite.

In the following, we will often use finite-support partial functions. Hence, to simplify the notation, by

$$f := \begin{cases} v_1 \mapsto r_1 \\ \quad\vdots \\ v_n \mapsto r_n \end{cases}$$

we will denote (by cases) any function f which assumes on input values $v_1, \dots, v_n$ output values $r_1, \dots, r_n$ and is otherwise undefined. Furthermore, if the support of f is just the singleton $\{v\}$, we will denote it by $f := v \mapsto r$.

A *total function* f from S to S' is a partial function such that, for all $x \in S$, there is some $y \in S'$ such that $f(x) = y$ ($supp(f) = S$). As in tradition, when we talk about a function we are referring to a total function, so, we will always say explicitly when a function is partial. To indicate that a function f from S to S' is total, we write $f\colon S \to S'$. Moreover, the set of all (total) functions from S to S' is denoted by $[S \to S']$.

A function $f\colon S \to S'$ is *injective* if and only if, for each $x, y \in S$, if $f(x) = f(y)$ then $x = y$. f is *surjective* if and only if, for each $x' \in S'$, there exists $x \in S$ such that $f(x) = x'$.

We denote by $f = g$ the extensional equality, i.e., for each $x \in S$, $f(x) = g(x)$.

Lambda Notation

Lambda notation provides a way of referring to functions without having to name them. Let $f\colon S \to S'$ be a function that for any element $x \in S$, gives a value $f(x)$ which is exactly described by expression E. We can express the function f as $\lambda x \in S.\, E$. Thus, $(\lambda x \in S.\, E) := \{(x, E[x]) \,|\, x \in S\}$ and so $\lambda x \in S.\, E$ is just an abbreviation for the set of input-output values determined by the expression $E[x]$. We use the lambda notation also to denote partial functions by allowing expressions in lambda-terms that are not always defined. Hence, a lambda expression $\lambda x \in S.\, E$ denotes a partial function $S \rightharpoonup S'$ which, on input $x \in S$, assumes the value $E[x] \in S'$ if the expression $E[x]$ is defined, and otherwise it is undefined.

Composing Relations and Functions

The *composition* of two relations $R : S \times S'$ and $Q : S' \times S''$ is a relation between S and S'' defined as $Q \circ R := \{(x, z) \in S \times S'' \,|\, y \in S', (x, y) \in R, (y, z) \in Q\}$. R^n is the relation $\underbrace{R \circ \cdots \circ R}_{n}$, i.e., $R^1 := R$ and (assuming R^n is defined) $R^{n+1} := R \circ R^n$. Each set S is associated with an identity function $Id_S := \{(x, x) \,|\, x \in S\}$, which is the neutral element of $\circ$. Thus we define $R^0 := Id_S$.

The *transitive and reflexive closure* R^* of a relation R on S is $R^* := \bigcup_{i \in \mathbf{N}} R^i$.

The *function composition* of $g: S \rightharpoonup S'$ and $f: S' \rightharpoonup S''$ is the partial function $f \circ g: S \rightharpoonup S''$, where $(f \circ g)(x) := f(g(x))$, if $g(x)$ (first) and $f(g(x))$ (then) are defined, and otherwise it is undefined. When it is clear from the context $\circ$ will be omitted.

A function $f : S \to S'$ is *bijective* if it has an *inverse* $g: S' \to S$, i.e., if and only if there exists a function g such that $g \circ f = Id_S$ and $f \circ g = Id_{S'}$. Then the sets S and S' are said to be in one-to-one correspondence. Any set in one-to-one correspondence with a subset of natural numbers $\mathbf{N}$ is said to be countable. Note that a function f is bijective if and only if it is injective and surjective.

Direct and Inverse Image of a Relation

We extend a relation $R: S \times S'$ to functions on subsets by taking $R(X) := \{y \in S' \mid \exists x \in X. (x, y) \in R\}$ for $X \subseteq S$. The set $R(X)$ is called the *direct image* of X under R. The *inverse image* of Y under R is defined as $R^{-1}(Y) := \{x \in S \mid \exists y \in Y. (x, y) \in R\}$ for $Y \subseteq S'$. Thus, if $f: S \rightharpoonup S'$ is a partial function, $X \subseteq S$ and $X' \subseteq S'$, we denote by $f(X)$ the *image* of X under f, i.e., $f(X) := \{f(x) \mid x \in X\}$ and by $f^{-1}(X')$ the *inverse image* of X' under f, i.e., $f^{-1}(X') := \{x \mid f(x) \in X'\}$.

Equivalence Relations and Congruences

An *equivalence relation* $\approx$ on a set S is a binary relation on S ($\approx: S \times S$) such that, for each $x, y, z \in S$,

$$x \, R \, x \qquad \text{(reflexivity)}$$
$$x \, R \, y \Longrightarrow y \, R \, x \qquad \text{(symmetricity)}$$
$$x \, R \, y, y \, R \, z \Longrightarrow x \, R \, z \qquad \text{(transitivity)}$$

The *equivalence class* of an element $x \in S$, with respect to $\approx$, is the subset $[x]_\approx := \{y \mid x \approx y\}$. When clear from the context, we abbreviate $[x]_\approx$ by $[x]$ and often abuse notation by letting the elements of a set denote their correspondent equivalence classes. The *quotient set* $S/_{\approx}$ of S modulo $\approx$ is the set of equivalence classes of elements in S (w.r.t. $\approx$).

An equivalence relation $\approx$ on S is a *congruence* w.r.t. a partial function $f : S^n \rightharpoonup S$ if and only if, for each pair of elements a_i, $b_i \in S$ such that $a_i \approx b_i$, if $f(a_1, \ldots, a_n)$ is defined then also $f(b_1, \ldots, b_n)$ is defined, and, furthermore, $f(a_1, \ldots, a_n) \approx f(b_1, \ldots, b_n)$. Then, we can define the partial function $f_\approx: (S/_{\approx})^n \rightharpoonup S/_{\approx}$ as $f_\approx([a_1]_\approx, \ldots, [a_n]_\approx) := [f(a_1, \ldots, a_n)]_\approx$, since, given $[a_1]_\approx, \ldots, [a_n]_\approx$, the class $[f(a_1, \ldots, a_n)]_\approx$ is uniquely determined independently of the choice of the representatives $a_1, \ldots, a_n$.

1.2 Domain Theory

In this section we present the (abstract) concepts of complete lattices, continuous functions and fixpoint theory, which are the standard tools for the definition of a denotational semantics.

1.2.1 Complete lattices and continuous functions

A binary relation $\leq$ on S ($\leq: S \times S$) is a *partial order* if, for each $x, y \in S$,

$$x \leq x \qquad \text{(reflexivity)}$$

$$x \le y, y \le x \Longrightarrow x = y \qquad \text{(antisymmetry)}$$

$$x \le y, y \le z \Longrightarrow x \le z \qquad \text{(transitivity)}$$

A *partially ordered set* (poset) $(S, \le)$ is a set S equipped with a partial order $\le$. A set S is *totally ordered* if it is partially ordered and, for each $x, y \in S$, $x \le y$ or $y \le x$. A *chain* is a (possibly empty) totally ordered subset of S.

A *preorder* is a binary relation which is reflexive and transitive. A preorder $\le$, on a set S, induces on S an equivalence relation $\approx$ defined as follows: for each $x, y \in S$,

$$x \approx y \Longleftrightarrow x \le y, y \le x.$$

Moreover, $\le$ induces on $S/_{\approx}$ the partial order $\le_{\approx}$ such that, for each $[x]_{\approx}, [y]_{\approx} \in S/_{\approx}$,

$$[x]_{\approx} \le_{\approx} [y]_{\approx} \Longleftrightarrow x \le y.$$

A binary relation $<$ is *strict* if and only if it is anti-reflexive (i.e., not $x < x$) and transitive.

Given a poset $(S, \le)$ and $X \subseteq S$, $y \in S$ is an *upper bound* for X if and only if, for each $x \in X$, $x \le y$. Moreover, $y \in S$ is the *least upper bound* (called also join) of X, if y is an upper bound of X and, for every upper bound y' of X, $y \le y'$. A least upper bound of X is often denoted by $lub_S X$ or by $\bigsqcup_S X$. We also write $\bigsqcup_S \{d_1, \ldots, d_n\}$ as $d_1 \sqcup_S \cdots \sqcup_S d_n$. Dually, an element $y \in S$ is a *lower bound* for X if and only if, for each $x \in X$, $y \le x$. Moreover, $y \in S$ is the *greatest lower bound* (called also meet) of X, if y is a lower bound of X and for every lower bound y' of X, $y' \le y$. A greatest lower bound of X is often denoted by $glb_S X$ or by $\bigsqcap_S X$. We also write $\bigsqcap_S \{d_1, \ldots, d_n\}$ as $d_1 \sqcap_S \cdots \sqcap_S d_n$. When it is clear from the context, the subscript S will be omitted. Moreover, $\bigsqcup \{D_i\}_{i \in I}$ and $\bigsqcap \{D_i\}_{i \in I}$ can be denoted by $\bigsqcup_{i \in I} D_i$ and $\bigsqcap_{i \in I} D_i$. It is easy to check that if *lub* and *glb* exist, then they are unique.

Complete Partial Orders and Lattices

A *direct set* is a poset in which any subset of two elements (and hence any finite subset) has an upper bound in the set. A *complete partial order* (*CPO*) S is a poset such that every chain D has the least upper bound (i.e., there exists $\bigsqcup D$). Notice that any set ordered by the identity relation forms a *CPO*, of course without a bottom element. Such *CPO*s are called discrete. We can add a bottom element to any poset $(S, \le)$ which does not have one (even to a poset which already has one). The new poset $S_\bot$ is obtained by adding a new element $\bot$ to S and by extending the ordering $\le$ as $\forall x \in S. \bot \le x$. If S is a discrete *CPO*, then $S_\bot$ is a *CPO* with bottom element, which is called flat.

A *complete join-semilattice* (respectively *complete meet-semilattice*) is a poset $(S, \le)$ such that, for every subset X of S, there exists $\bigsqcup X$ (respectively $\bigsqcap X$).

A *complete lattice* is a poset $(S, \le)$ such that, for every subset X of S, there exists $\bigsqcup X$ and $\bigsqcap X$. Let $\top$ denote the *top element* $\bigsqcup S = \bigsqcap \varnothing$ and $\bot$ denote the *bottom element* $\bigsqcap S = \bigsqcup \varnothing$ of S. The elements of a complete lattice can be seen as points of information, and the ordering as an approximation relation between them. Thus, $x \le y$ means x approximates y (or, x has less or the same information as y) and so $\bot$ is the point of least information. It is easy to check that, for any set S, $\wp(S)$ under the subset ordering $\subseteq$ is a complete lattice, where the least upper bound is the union, the greatest lower bound is the intersection, the top element is S, and the bottom element is $\varnothing$. Also $(\wp(S))_\bot$ is a complete lattice.

Given a complete lattice $(L, \leq)$, the set of all partial functions $F = [S \rightharpoonup L]$ inherits the complete lattice structure of L. Let simply define $f \leq g := \forall x \in S. f(x) \leq g(x)$, $(f \sqcup g)(x) := f(x) \sqcup g(x)$, $(f \sqcap g)(x) := f(x) \sqcap g(x)$, $\bot_F := \lambda x \in S. \bot_L$ and $\top_F := \lambda x \in S. \top_L$.

Continuous and Additive Functions

Let $(L, \leq)$ and $(M, \sqsubseteq)$ be (complete) lattices. A function $f\colon L \to M$ is *monotonic* if and only if

$$\forall x, y \in L. x \leq y \Longrightarrow f(x) \sqsubseteq f(y).$$

Moreover, f is *continuous* if and only if, for each non-empty chain $D \subseteq L$,

$$f(\bigsqcup_L D) = \bigsqcup_M f(D).$$

Every continuous function is also monotonic, since $x \leq y$ implies $f(\bigsqcup_L\{x,y\}) = f(y)$, by continuity $\bigsqcup_M\{f(x), f(y)\} = f(\bigsqcup_L\{x,y\})$, which implies that $f(x) \sqsubseteq f(y)$, since $f(x) \sqsubseteq \bigsqcup_M\{f(x), f(y)\}$ and we have already seen that $f(\bigsqcup_L\{x,y\}) = f(y)$.

Complete partial orders correspond to types of data (which can be used as input or output to a computation) and computable functions are modeled as continuous functions between them.

A partial function $f\colon S \rightharpoonup S'$ is *additive* if and only if the previous continuity condition is satisfied for each non-empty set. Hence, every additive function is also continuous. Dually we define *co-continuity* and *co-additivity*, by using $\sqcap$ instead of $\sqcup$.

It can be proven that the composition of monotonic, continuous or additive functions is, respectively, monotonic, continuous or additive.

The mathematical way of expressing that two structures are "essentially the same" is given through the concept of isomorphism. A continuous function $f\colon D \to E$ between *CPO*s D and E is said to be an *isomorphism* if there is a continuous function $g\colon E \to D$ such that $g \circ f = Id_D$ and $f \circ g = Id_E$ (f and g are mutual inverses). It follows from the definition, that two isomorphic *CPO*s are essentially the same but for a renaming of elements. It can be proven that a function $f\colon D \to E$ is an isomorphism if and only if f is bijective and, for all $x, y \in D$, $x \leq_D y \Longleftrightarrow f(x) \leq_E f(y)$.

Function Space

Let D, E be *CPO*s, the function space $[D \to E]$ consists of continuous functions $f\colon D \to E$ ordered pointwise by $f \sqsubseteq g \Longleftrightarrow \forall d \in D. f(d) \sqsubseteq g(d)$. Note that, if E has a bottom element $\bot_E$, also $[D \to E]$ has a bottom element: the constantly $\bot_E$ function $\bot_{[D \to E]} := \lambda d \in D. \bot_E$. Least upper bounds of chains of functions are given pointwise, i.e., a chain of functions $f_0 \sqsubseteq f_1 \sqsubseteq \ldots \sqsubseteq f_n \sqsubseteq \ldots$ has *lub* $\bigsqcup_{[D \to E]} f_n := \lambda d \in D. \bigsqcup_E\{f_n(d)\}_{n \in \mathbf{N}}$. It is easy to see that $[D \to E]$ forms a complete partial order with the order relation $\sqsubseteq$.

Partial functions $L \rightharpoonup D$ are in one-to-one correspondence with (total) functions $L \to D_\bot$, and, in this case, any total function is continuous. The inclusion order between partial functions corresponds to the "pointwise order" $f \sqsubseteq g \Longleftrightarrow \forall \sigma \in L. f(\sigma) \sqsubseteq g(\sigma)$ between functions $L \to D_\bot$. Since partial functions can be undefined on some input, to keep the correctness of the "pointwise order", we assume that, for each set S, $\aleph \subseteq S$, $\aleph \cup S = S$ and $\varnothing \notin \aleph$ (see Section 1.1.2).

1.2.2 Fixpoint Theory

Given a poset $(S, \leq)$ and a function $f\colon S \to S$, a *fixpoint* of f is an element $x \in S$ such that $f(x) = x$. A *pre-fixpoint* of f is an element $x \in S$ such that $f(x) \leq x$ and, dually, a *post-fixpoint* of f is an element $x \in S$ such that $x \leq f(x)$. Moreover, we say that $x \in S$ is the *least fixpoint of* f (denoted by $\mathit{lfp}\, f$) if and only if x is a fixpoint of f and for all fixpoints y of f, $x \leq y$. Dually, we define the *greatest fixpoint* (denoted by $\mathit{gfp}\, f$).

The fundamental theorem of Knaster-Tarski states that the set of fixpoints of a monotonic function f is a complete lattice.

Theorem 1.2.1 (Knaster-Tarski Fixpoint theorem [114]) *A monotonic function f on a complete lattice $(L, \leq)$ has the least fixpoint and the greatest fixpoint. Moreover,*

$$\begin{aligned} \mathit{lfp}(f) &= \bigsqcap\{x \mid f(x) \leq x\} = \bigsqcap\{x \mid x = f(x)\} \\ \mathit{gfp}(f) &= \bigsqcup\{x \mid x \leq f(x)\} = \bigsqcup\{x \mid x = f(x)\}. \end{aligned}$$

The Knaster-Tarski Theorem is important because it applies to any monotone function on a complete lattice. However, most of the time we will be concerned with least fixpoints of continuous functions which we will construct by the techniques of the previous section, as least upper bounds of chains in a complete lattice. Therefore, it is useful to state some more notations and results on fixpoints of continuous functions defined on (complete) lattices.

First of all, we have to introduce the notion of *ordinal*. We assume that an ordinal is a set where every element of an ordinal is still an ordinal and the class of ordinals is ordered by membership relation ($\alpha < \beta$ means $\alpha \in \beta$). Consequently, every ordinal coincides with the set of all smaller ordinals. The least ordinals are 0, $1 := \{0\}$, $2 := \{0, \{0\}\}$, *etc.*. Intuitively, the class of ordinals is the transfinite sequence $0 < 1 < 2 < \ldots < \omega < \omega + 1 < \ldots < \omega + \omega < \ldots < \omega^\omega$, *etc.*. Ordinals will be often denoted by Greek letters. An ordinal γ is a *limit ordinal* if it is neither 0 nor the successor of an ordinal; so, if $\beta < \gamma$, then there exists σ such that $\beta < \sigma < \gamma$. The first limit ordinal, which is equipotent with the set of natural numbers, is denoted (by an abuse of notation) by ω. Often, in the definitions of *CPO* and of continuity, directed sets are used instead of chains. It is possible to show that if the set S is denumerable, then the definitions are equivalent.

The ordinal powers of a monotonic function $T\colon S \to S$ on a *CPO* S are defined as

$$T{\uparrow}\alpha(x) := \begin{cases} x & \text{if } \alpha = 0 \\ T(T{\uparrow}(\alpha - 1)(x)) & \text{if } \alpha \text{ is a successor ordinal} \\ \bigsqcup\{T{\uparrow}\beta(x) \mid \beta < \alpha\} & \text{if } \alpha \text{ is a limit ordinal.} \end{cases}$$

In the following, we will use the standard notation $T{\uparrow}\alpha := T{\uparrow}\alpha(\bot)$, where $\bot$ is the bottom of S. In particular, $T{\uparrow}\omega := \bigsqcup_{n<\omega} T{\uparrow}n$, $T{\uparrow}n + 1 := T(T{\uparrow}n)$, for $n < \omega$, and $T{\uparrow}0 := \bot$, where $\bigsqcup$ is the *lub* operation of S. Sometimes, $T{\uparrow}\alpha(x)$ may be denoted simply by $T^\alpha(x)$.

The next important result is usually attributed to Kleene and gives an explicit construction of the least fixpoint of a continuous function f on a *CPO* D.

Theorem 1.2.2 (Kleene Fixpoint theorem) *Let $f: D \to D$ be a continuous function on a CPO D, and $d \in D$ be a pre-fixpoint of f. Then $\bigsqcup\{f\uparrow n(d) \mid n \leq \omega\}$ is the least fixpoint of f greater than d. In particular $f\uparrow\omega$ is the least pre-fixpoint and least fixpoint of f.*

Each *CPO* D with bottom $\perp$ is associated with a fixpoint operator $\mathit{fix}: [D \to D] \to D$, $\mathit{fix} := \bigsqcup_{n<\omega}(\lambda f.\, f^n(\perp))$, i.e., *fix* is the least upper bound of the chain of the functions $\lambda f.\perp \sqsubseteq \lambda f.\, f(\perp) \sqsubseteq \lambda f.\, f(f(\perp)) \sqsubseteq \ldots$, where each of these is continuous and, therefore, it is an element of the *CPO* $[[D \to D] \to D]$.

1.3 Abstract Interpretation

Abstract interpretation [31, 33] is a general approximation theory for reasoning about semantic properties of discrete dynamic systems. The *abstract semantics* is an approximation of the concrete one, where exact (concrete) values are replaced by approximated (abstract) values, which model some interesting properties on the program execution. In this section we present the theoretical foundation and the principal results of abstract interpretation theory which will be used in this thesis.

1.3.1 Closures on complete lattices

Closures play a fundamental role in semantics and approximation theory [33]. In the following, we recall some basic notions on closure theory. For a more complete treatment of the subject see [14, 32]. A *closure operator* on a complete lattice $(L, \leq)$ is an operator $\rho: L \to L$ such that, for each $x, y \in L$,

$$
\begin{array}{ll}
x \leq \rho(x) & \text{(extensivity)} \\
x \leq y \Longrightarrow \rho(x) \leq \rho(y) & \text{(monotonicity)} \\
\rho(\rho(x)) = \rho(x) & \text{(idempotence)}
\end{array}
$$

Let $(L, \leq)$ be a complete lattice, in the following, we enumerate some basic properties of closure operators on L. Let ρ be an upper closure operator on $(L, \leq)$.

- For all $x \in L$, the set $\{y \in \rho(L) \,|\, x \leq y\}$ is not empty and $\rho(x)$ is the least element.
- The image $R := \rho(L)$ of L by ρ is a complete lattice $(R, \leq)$, such that $\bigsqcup_R(X) = \rho(\bigsqcup_L(X))$ and $\bigsqcap_R(X) = \bigsqcap_L(X)$.
- ρ is a *quasi-complete-join-morphism*, i.e., for each $X \subseteq L$, $\rho(\bigsqcup(X)) = \rho(\bigsqcup(\rho(X)))$.
- Let $R \subseteq L$ and $\rho: L \to R$ such that, for any $x \in L$, $\rho(x)$ is the least element in $\{y \in R \,|\, x \leq y\}$. Then ρ is an upper closure operator on $(L, \leq)$ and $R := \rho(L)$.
- Let $\mathit{uco}(L)$ be the set of all upper closure operators on L. Then $(\mathit{uco}(L), \leq)$ is a complete lattice, where $\leq$ is defined as follows. For each $\rho, \rho' \in \mathit{uco}(L)$,

  $$\rho \leq \rho' \iff \forall x \in L.\, \rho(x) \leq \rho'(x).$$

1.3.2 Galois Connections

Abstract interpretation theory requires the two semantics (concrete and abstract) to be defined on domains which are partially ordered sets. $(\mathbf{C}, \sqsubseteq)$ (the concrete domain) is the domain of the concrete semantics, while $(\mathbf{A}, \leq)$ (the abstract domain) is the domain of the abstract semantics. The partial order relations reflect an approximation relation.

The concrete and the abstract interpretation are related by a pair of functions, the *abstraction* α and the *concretization* γ, which form a Galois Connection. This notion has been introduced in [91] to discuss a general type of correspondence between structures occurring in a great variety of mathematical theories.

Galois Connections can be defined on partially ordered sets. However, in this thesis we restrict our attention to complete lattices since they meet stronger properties.

In approximation theory, a partial order specifies the precision degree of any element in a poset. Thus, it is obvious to assume that, if α is a mapping associating an abstract object in $(\mathbf{A}, \leq)$ to any concrete element x in $(\mathbf{C}, \sqsubseteq)$, the following holds: if $\alpha(x) \leq y$, then y is also a correct, although less precise, abstract approximation of x. The same argument holds if $x \sqsubseteq \gamma(y)$. Then y is also a correct approximation of x, although x provides more accurate information than $\gamma(y)$. This gives rise to the following formal definition.

Definition 1.3.1 (Galois Connection) *Let* $(\mathbf{C}, \sqsubseteq, \bigsqcup, \bigsqcap, \top, \bot)$ *and* $(\mathbf{A}, \leq, \bigvee, \bigwedge, \top, \bot)$ *be two complete lattices. A* Galois Connection $(\mathbf{C}, \sqsubseteq) \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} (\mathbf{A}, \leq)$ *is a pair of maps* $\alpha : \mathbf{C} \to \mathbf{A}$ *and* $\gamma : \mathbf{A} \to \mathbf{C}$ *such that, for each* $x \in \mathbf{C}$ *and* $y \in \mathbf{A}$*,*

$$\alpha(x) \leq y \iff x \sqsubseteq \gamma(y) \tag{1.3.1}$$

Moreover, a Galois Insertion *(of* $\mathbf{A}$ *in* $\mathbf{C}$*)* $(\mathbf{C}, \sqsubseteq) \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} (\mathbf{A}, \leq)$ *is a Galois connection where* α *is surjective.*

An equivalent definition of Galois Connection is a pair of maps $\alpha : \mathbf{C} \to \mathbf{A}$ and $\gamma : \mathbf{A} \to \mathbf{C}$ such that:

Monotonicity. The maps α and γ are monotonic.
Extensivity. For each $x \in \mathbf{C}$, $x \sqsubseteq (\gamma \circ \alpha)(x)$.
Reductivity. For each $y \in \mathbf{A}$, $(\alpha \circ \gamma)(y) \leq y$.

When, in a Galois Connection $\overset{\gamma}{\underset{\alpha}{\leftrightarrows}}$, γ is not injective, several distinct elements of the abstract domain $(\mathbf{A}, \leq)$ have the same meaning (by γ). This is usually considered useless [33]; in this situation a Galois Insertion can always be forced by considering a more concise abstract domain $(\mathbf{A}/_{\overset{\gamma}{\approx}}, \leq/_{\overset{\gamma}{\approx}})$, such that for each $x, y \in A$. $x \overset{\gamma}{\approx} y \iff \gamma(x) = \gamma(y)$.

The following basic properties are satisfied by any Galois Connection:

1. γ is injective if and only if α is surjective if and only if $\alpha \circ \gamma = id_{\mathbf{A}}$;
2. α is injective if and only if γ is surjective if and only if $\gamma \circ \alpha = id_{\mathbf{C}}$;
3. α is additive ($\alpha(\bigsqcup X) = \bigvee \alpha(X)$) and γ is co-additive ($\gamma(\bigwedge Y) = \bigsqcap \gamma(Y)$);
4. α and γ uniquely determine each other:

$$\gamma(y) = \bigsqcup \{x \in \mathbf{C} \mid \alpha(x) \leq y\}, \qquad \alpha(x) = \bigwedge \{y \in \mathbf{A} \mid x \sqsubseteq \gamma(y)\};$$

5. $\gamma \circ \alpha$ is an upper closure operator in $(\mathbf{C}, \sqsubseteq)$.

6. α is an isomorphism from $(\gamma\alpha)(\mathbf{C})$ to $\mathbf{A}$, having γ as its inverse.

It follows immediately from Property 1 that in the case of Galois Insertions, the Property Reductivity in the alternative definition of Galois Connection is equivalent to state that $(\alpha \circ \gamma) = id_{\mathbf{A}}$ for each $y \in \mathbf{A}$.

Because of Property 4 the map α (respectively γ) is called the *lower* (respectively *upper*) *adjoint*. Properties 5 and 6 characterize the ability of Galois Connections to formalize the notion of "machine-representable" abstractions. An abstract domain is isomorphic (up to representation) to an upper closure operator of the concrete domain of the computation $\mathbf{C}$. Thus, in principle, we can handle abstract computations as concrete computations on the complete lattice which is the image of the upper closure operator $\gamma \circ \alpha$. However, machine representable abstractions often result to be more intuitive and provide better experimental results in efficient implementations.

A straightforward consequence of the latter observation is that abstract interpretations can be formalized in a hierarchical framework. Abstract domains can be partially ordered using the ordering on the corresponding closure operators on $\mathbf{C}$. The lattice of abstract interpretations of $\mathbf{C}$ is, then, the lattice of upper closure operators over $\mathbf{C}$. As observed in [91], the composition of upper closure operators is not (in general) an upper closure operator. However, an abstract domain can be designed by successive approximations. Let ρ be an upper closure operator on $(\mathbf{C}, \sqsubseteq)$ and η be an upper closure operator on $\rho(\mathbf{C})$. Then $\eta \circ \rho$ is an upper closure operator on $(\mathbf{C}, \sqsubseteq)$.

In view of the compositional design of abstract interpretations we have that the composition of Galois Connections is a Galois Connection. Several techniques can be used to systematically derive new abstract interpretations from a given set of abstract domains [33, 34]. We do not address these techniques because they are outside the scope of this thesis.

1.3.3 Abstract semantics, correctness and precision

In program analysis based on abstract interpretation, we compute an abstract (fixpoint) semantics in order to reason about the program behavior. Given a concrete semantics and a Galois Insertion between the concrete and the abstract domain, we want to define an abstract semantics. The concrete semantics of a program D is usually formalized as the least fixpoint of a continuous semantics evaluation function, $\mathcal{D}[\![D]\!] \colon \mathbf{C} \to \mathbf{C}$, on the concrete domain $(\mathbf{C}, \sqsubseteq)$. The class of program properties we want to consider is formalized by a complete lattice $(\mathbf{A}, \leq)$. Concrete and abstract domains are related by a Galois Insertion $(\mathbf{C}, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbf{A}, \leq)$.

An abstract semantic function $\mathcal{D}^a[\![D]\!] \colon \mathbf{A} \to \mathbf{A}$ is *correct* if $\forall x \in \mathbf{C}$

$$\mathcal{D}[\![D]\!](x) \sqsubseteq \gamma(\mathcal{D}^a[\![D]\!](\alpha(x))).$$

The resulting abstract semantics $lfp_{\mathbf{A}}(\mathcal{D}^a[\![D]\!])$ is a correct approximation of the concrete one by construction, i.e., $\alpha(lfp_{\mathbf{C}}(\mathcal{D}[\![D]\!])) \leq lfp_{\mathbf{A}}(\mathcal{D}^a[\![D]\!])$.

Moreover, we can systematically derive from $\mathcal{D}[\![D]\!]$, α and γ a correct abstract semantic evaluation function simply as $\mathcal{D}^\alpha[\![D]\!] := \alpha \circ \mathcal{D}[\![D]\!] \circ \gamma$. $\mathcal{D}^\alpha[\![D]\!]$ is shown to be the *most precise* and *correct* abstract counterpart of $\mathcal{D}[\![D]\!]$, as for any correct $\mathcal{D}^{a}[\![D]\!]$, $\mathcal{D}^{\alpha}[\![D]\!] \leq \mathcal{D}^{a}[\![D]\!]$. Thus $\mathcal{D}^{\alpha}[\![D]\!]$ is called the *optimal abstract version* of $\mathcal{D}[\![D]\!]$.

The abstract semantics $lfp_{\mathbf{A}}(\mathcal{D}^{\alpha}[\![D]\!])$ models a safe approximation of the property of interest: if the property is verified in $lfp_{\mathbf{A}}(\mathcal{D}^{\alpha}[\![D]\!])$ it will also be verified in $lfp_{\mathbf{C}}(\mathcal{D}[\![D]\!])$. An analysis method based on the computation of the abstract semantics $lfp_{\mathbf{A}}(\mathcal{D}^{\alpha}[\![D]\!])$ is effective only if the least fixpoint is reached in finitely many iterations, i.e., if the abstract domain is Nötherian. If this is not the case, *widening operators* can be used to ensure the termination. Widening operators [36] give an upper approximation of the least fixpoint and guarantee termination by introducing further approximation.

The framework of abstract interpretation is useful to study hierarchies of semantics and to reconstruct data-flow analysis methods and type systems. It can also be used to systematically derive "optimal" abstract semantics from the abstract domain and in this way design efficient verification and analysis methods.

Furthermore, the systematic design aspect can be pushed forward, by using suitable abstract domain design methodologies (e.g. domain refinements) [55, 62, 64], which allow one to systematically improve the precision of the domain.

1.3.4 Correctness and precision of compositional semantics

Usually, $\mathcal{D}^{\alpha}[\![D]\!]$ is defined as the composition of "primitive" operators. Let $f : \mathbf{C}^n \to \mathbf{C}$ be one of these operators and assume that $\tilde{f}$ is its abstract counterpart. Then $\tilde{f}$ is *(locally) correct* w.r.t. f if $\forall x_1, \dots, x_n \in \mathbf{C}.\, f(x_1, \dots, x_n) \sqsubseteq \gamma(\tilde{f}(\alpha(x_1), \dots, \alpha(x_n)))$. By replacing all concrete f with the abstract $\tilde{f}$ in the formal definition of $\mathcal{D}^{\alpha}[\![D]\!]$, we obtain the definition of an abstract operator $\mathcal{D}^{a}[\![D]\!]$. The local correctness of all the primitive operators implies the global correctness (the correctness of $\mathcal{D}^{a}[\![D]\!]$). Hence, we can define an abstract semantics by defining locally correct abstract primitive semantic functions. According to the theory, for each operator f, there exists an optimal (most precise) locally correct abstract operator $\tilde{f}$ defined as $\tilde{f}(y_1, \dots, y_n) = \alpha(f(\gamma(y_1), \dots, \gamma(y_n)))$. However, the composition of optimal operators is not necessarily optimal.

The abstract operator $\tilde{f}$ is *precise* if $\forall x_1, \dots, x_n \in \mathbf{C}$

$$\alpha(f(x_1, \dots, x_n)) = \tilde{f}(\alpha(x_1), \dots, \alpha(x_n))$$

which is equivalent to

$$\alpha(f(x_1, \dots, x_n)) = \alpha(f((\gamma \circ \alpha)(x_1), \dots, (\gamma \circ \alpha)(x_n))).$$

Hence the precision of an optimal abstract operator can be reformulated in terms of properties of α, γ and the corresponding concrete operator.

There is not currently an agreement on a name for what we call precision. For instance: [63] calls it full-completeness; [37, 85, 101, 112] use the term *completeness*; while [39] use the term *optimality* for the same notion. We prefer to use the term precision, since completeness may be confused with the completeness of a semantics.

Note that if $\bigsqcup$ is the *lub* operation over $(\mathbf{C}, \sqsubseteq)$ and $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$ is a Galois Insertion then $\widetilde{\bigsqcup} = \alpha \circ \bigsqcup \circ \gamma$ is the *lub* of $(\mathbf{A}, \leq)$ and is precise, i.e., $\widetilde{\bigsqcup} \circ \alpha = \alpha \circ \bigsqcup$ (which is equivalent to $\alpha \circ \bigsqcup = \alpha \circ \bigsqcup \circ \gamma \circ \alpha$).

1.4 Constraint Systems

A (simple) constraint system [110] is a system of partial information where constraints can be viewed as elementary tokens which assert some partial information on the current state. Constraints are related by means of an entailment relation $\vdash$: given two constraints c_1 and c_2, $c_1 \vdash c_2$ holds if and only if c_1 "contains " more information than c_2 (i.e., c_1 is a more restrictive constraint than c_2).

The notion of constraint systems able to handle queries with existential quantified variables was first described in [104]. A more elegant formalization, the cylindric constraint systems, was introduced in [109].

A cylindric constraint system [109, 110] extends the notion of simple constraint system [109, 110] by borrowing from cylindric algebras the notions of cylindrification operator and diagonal element [67]. However, since we are dealing with *tccp*, in this thesis we prefer to use the formalization of [43].

Definition 1.4.1 (Cylindric constraint system [43]) *A cylindric constraint system is an algebraic structure of the form:*

$$\mathbf{C} = \langle \mathcal{C}, \leq, \otimes, \oplus, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$$

such that:

1. $\langle \mathcal{C}, \leq, \otimes, \oplus, \mathit{true}, \mathit{false} \rangle$ *is a complete lattice where, following the standard notation,* $\otimes$ *is the lub operator,* $\oplus$ *is the corresponding glb operator, and true and false are, respectively, the least and the greatest elements of* $\mathcal{C}$. *We often use the inverse order* $\vdash$ *(the* entailment *relation) instead of* $\leq$ *over constraints. Formally* $\forall c, d \in \mathcal{C}$ $c \leq d \Leftrightarrow d \vdash c$.

2. *Var is a denumerable set of variables.*

3. *For each element* $x \in \mathit{Var}$, *a function (also called cylindric operator)* $\exists_x{:}\mathcal{C} \to \mathcal{C}$ *is defined such that, for any* $c, d \in \mathcal{C}$ *the following axioms hold:*

   (a) $c \vdash \exists_x c$

   (b) *if* $c \vdash d$ *then* $\exists_x c \vdash \exists_x d$

   (c) $\exists_x(c \otimes \exists_x d) = \exists_x c \otimes \exists_x d$

   (d) $\exists_x(\exists_y c) = \exists_y(\exists_x c)$

4. *To model parameter passing,* diagonal elements *are added to the primitive constraints. For all x, y ranging over Var, the constraint* d_{xy} *which satisfies the following axioms is added.*

   (a) $\mathit{true} \vdash d_{xx}$

   (b) *if* $z \neq x, y$ *then* $d_{xy} = \exists_z(d_{xz} \otimes d_{zy})$

   (c) *if* $x \neq y$ *then* $\exists_{xy}(c \otimes d_{xy}) \vdash c$.

Here the *cylindrification* (or *hiding*) operator is defined in terms of a general notion of existential quantifier. It is used to project away information about the considered variable in order to make it local to the constraint and hide it to the context. *Diagonal elements* are used to perform variable renaming. For instance the constraint $\exists_x(d_{xy} \otimes c)$ can be interpreted as the constraint $c[y \setminus x]$ where all the free occurrences of x are replaced by y.

We abuse in notation by denoting as $\exists_x C$ the extension of the existential quantification to the set of constraints C. Formally, given $C \subseteq \mathcal{C}$, we define $\exists_x C := \{\exists_x c \mid c \in C\}$.

We can find in the literature several examples of cylindric constraint systems that are useful when modeling data structures, logic programs or other specific domains [110, 41, 42, 7]. Let us introduce some examples that we will use throughout the thesis.

The Herbrand constraint system is well-known in the literature [46, 41]:

Cylindric Constraint System 1.4.2 *Given an alphabet consisting of variables $x, y, \dots \in$ Var, functions symbols $f, g, \dots$, constant symbols (i.e., function symbols of arity 0) $a, b, \dots$ and the equality predicate =, the Herbrand cylindric constraint system is the structure* $\mathbf{H} := \langle \wp(\mathcal{H}), \preceq_{\mathcal{H}}, \cup, \cap, \mathcal{H}, \varnothing, \mathit{Var}, \exists \rangle$ *where:*

1. *the elements of $\mathcal{H}$ are equations $t = u$ where t and u are terms of the alphabet;*
2. *given $H_1, H_2 \in \wp(\mathcal{H})$, $x \in \mathit{Var}$, t, u, v, t_i, u_i, v_j terms of the alphabet with $i = 1 \dots n$ and $j = 1 \dots m$, and f and g distinct function symbols of arity n and m respectively, the relation $\preceq_{\mathcal{H}}$ satisfies the following conditions:*

   H1 $H_1 \subseteq H_2$ *implies* $H_1 \preceq_{\mathcal{H}} H_2$

   H2 $\{t = t\} \preceq_{\mathcal{H}} \varnothing$

   H3 $\{t = u\} \preceq_{\mathcal{H}} \{u = t\}$

   H4 $\{t = u\} \preceq_{\mathcal{H}} \{t = v, v = u\}$

   H5 $\{f(t_1, \dots, t_n) = f(u_1, \dots, u_n)\} \preceq_{\mathcal{H}} \{t_1 = u_1, \dots, t_n = u_n\}$

   H6 *for* $i = 1 \dots n$, $\{t_i = u_i\} \preceq_{\mathcal{H}} \{f(t_1, \dots, t_n) = f(u_1, \dots, u_n)\}$

   H7 *for every set* $H \in \wp(\mathcal{H})$, $H \preceq_{\mathcal{H}} \{f(t_1, \dots, t_n) = g(v_1, \dots, v_m)\}$

   H8 *if x occurs in t and x is syntactically different from t, for every set $H \in \wp(\mathcal{H})$,* $H \preceq_{\mathcal{H}} \{x = t\}$;
3. *$\exists_x$ represent the existential quantifier.*

Condition H1 states the consistency of $\preceq_{\mathcal{H}}$ w.r.t. the set inclusion. Conditions H2, H3 and H4 stands for the standard axioms of reflexivity, symmetry and transitivity, respectively, while condition H5 corresponds to the notion of substitutivity. Finally, conditions H6, H7 and H8 model the so-called free-equality axioms ([19]) which enforce the interpretation of the equality = as syntactical identity. In particular, H7 and H8 express the fact that $f(t_1, \dots, t_n) = g(v_1, \dots, v_m)$ and $x = t$ for x occurring in t are equivalent to $\mathcal{H}$.

The following examples are taken from [46, 41]. Figure 1.4 shows graphically an example of Herbrand cylindric constraint system with the variable symbols x and y and the constant symbols a and b. Figure 1.4 represents an example of Herbrand cylindric constraint system, where the alphabet is formed by a free variable x, a constant symbol a and a monadic function symbol f. Here $x = f^{\omega}$ represent the limit of the chain $\{\exists_y(x = f^i(y))\}_i$.

The following constraint system allows one to model linear and equality constraints.

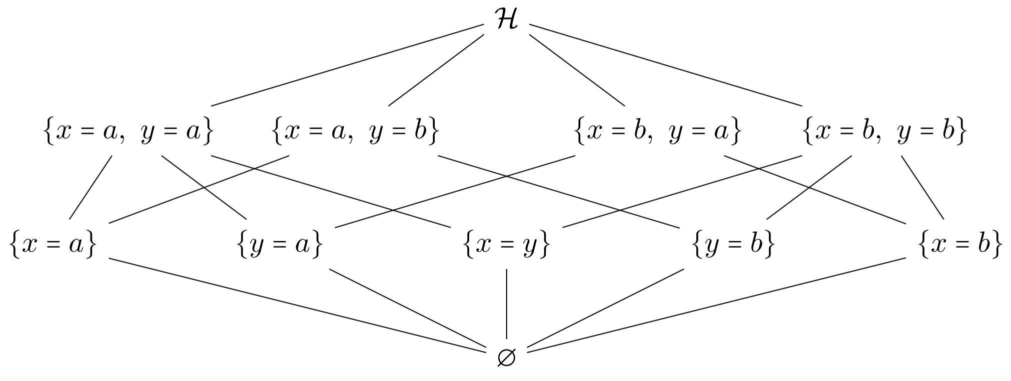


Figure 1.1: The Herbrand cylindric constraint system for x, y, a and b

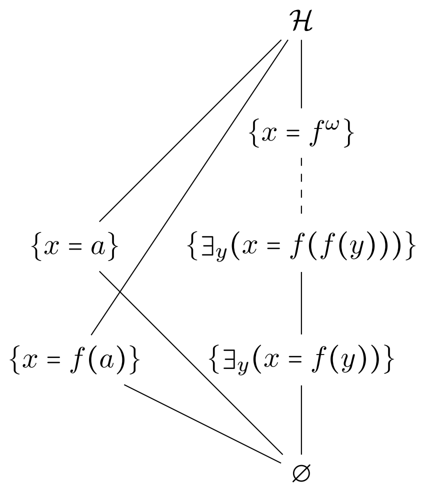


Figure 1.2: The Herbrand cylindric constraint system for x, a and f

Cylindric Constraint System 1.4.3 *The domain of constraints $\mathcal{L}$ is formed by taking equivalence classes, modulo logical equivalence $\Leftrightarrow$, of finite conjunctions of either linear disequalities (strict and not) or equalities over* $\mathbf{Z}$ *and* $Var = \{x, y, \dots\}$ *(e.g.* $x > 4$*,* $y \geq 10 \wedge w < -3$*, ...). The entailment relation is the implication* $\Rightarrow$ *(thus, the order of the lattice is* $\Leftarrow$*). The lub is the conjunction* $\wedge$*, the glb is the disjunction* $\vee$ *and* $\exists_x$ *is the operation which removes (after information has been propagated within a constraint) all conjuncts referring to variable* x *(e.g.* $\exists_x(x = y \wedge x > 3) = y > 3$*). It can be easily verified that* $\mathbf{L} := \langle \mathcal{L}, \Leftarrow, \wedge, \vee, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$ *is a cylindric constraint system.*

1.5 Linear Temporal Logic

Classical propositional and first-order logics can be used to express properties about program states. Each formula represents a set of states that satisfy it, thus, they can be used to express either an initial or final condition, or an invariant of a program. These logics are static, in the sense that they can represent a collection of states but not the dynamic evolution between them during the execution of a program.

Modal logics (see [73]) extend classical logics by including operators expressing modality. In this way it is possible to describe the relations between different states during the execution.

Among this family we distinguish *temporal logics* which are based on temporal modalities. These logics are suitable to express properties about concurrent and reactive systems where we are not interested only in the initial and final state (which it is not assured to exist) but also in the evolution of the state during the execution.

Linear temporal logic (in short LTL) [78] is an instance of temporal logic. LTL is defined on top of a static logic L, which can be a classic propositional or first-order logic, or either a constraint based logic (see [42, 88, 116]). An LTL formula is interpreted over a model, which is an *infinite* sequence of states $\sigma = s_1 \cdot s_2 \cdot s_3 \dots$. In the following we write σ^i for the sub-sequence $s_i \cdot s_{i+1} \dots$ and $\sigma(i)$ for the i-th state s_i.

Definition 1.5.1 (LTL formulas) *Let* ψ *be a formula in the underlying static logic* L*. An* LTL *formula has the following syntax:*

$$\phi ::= \dot{\mathit{true}} \mid \dot{\mathit{false}} \mid \psi \mid \dot{\neg}\,\phi \mid \phi \mathrel{\dot{\wedge}} \phi \mid \bigcirc\,\phi \mid \phi\, \mathcal{U}\, \phi.$$

The dot on top of the logic connectives is used in this thesis to avoid confusion with the operators of the constraint system.

The formulas $\dot{\mathit{true}}$, $\dot{\mathit{false}}$, $\dot{\neg}\,\phi$, and $\phi_1 \mathrel{\dot{\wedge}} \phi_2$ have the classical logical meaning. The atomic formula ψ of the logic L express a property about the current state. $\bigcirc$ is the *next* operator, i.e., the formula $\bigcirc\,\phi$ holds at position i if and only if ϕ holds at the next position $i+1$. The *until* formula $\phi_1\, \mathcal{U}\, \phi_2$ states that ϕ_2 eventually holds and in all previous instants ϕ_1 holds.

In this thesis, we will use $\phi_1 \mathrel{\dot{\vee}} \phi_2$ as a shorthand for $\dot{\neg}(\dot{\neg}\,\phi_1 \mathrel{\dot{\wedge}} \dot{\neg}\,\phi_2)$; $\phi_1 \mathrel{\dot{\rightarrow}} \phi_2$ for $\dot{\neg}\,\phi_1 \mathrel{\dot{\vee}} \phi_2$; $\Diamond\,\phi$ for $\dot{\mathit{true}}\, \mathcal{U}\, \phi$ and $\Box\,\phi$ for $\dot{\neg}\,\Diamond\,\dot{\neg}\,\phi$. $\Diamond$ is called *eventually* operator and the formula $\Diamond\,\phi$ holds at position i if and only if it exists $j > i$ such that ϕ holds at position j. The formula $\Box\,\phi$ is read *always* ϕ and states that ϕ holds from now on.

We write $s \vDash_{\mathsf{L}} \psi$ to denote that the state s models the formula ψ in the underlying logic L:

Definition 1.5.2 *For each $\phi, \phi_1, \phi_2 \in \mathsf{LTL}$, $\psi \in \mathsf{L}$ and σ infinite sequence of states, the LTL satisfaction relation $\models$ is defined as:*

$$\sigma \models \dot{true} \text{ and } \sigma \not\models \dot{false} \tag{1.5.1a}$$

$$\sigma \models \psi \qquad \text{iff } \sigma(1) \models_{\mathsf{L}} \psi \tag{1.5.1b}$$

$$s \models \dot{\neg}\, \phi \qquad \text{iff } \sigma \not\models \phi \tag{1.5.1c}$$

$$\sigma \models \phi_1 \dot{\wedge} \phi_2 \qquad \text{iff } \sigma \models \phi_1 \text{ and } \sigma \models \phi_2 \tag{1.5.1d}$$

$$\sigma \models \bigcirc \phi \qquad \text{iff } \sigma^1 \models \phi \tag{1.5.1e}$$

$$\sigma \models \phi_1 \,\mathcal{U}\, \phi_2 \qquad \text{iff } \exists i \geq 1.\, \sigma^i \models \phi_2 \text{ and } \forall j < i.\, \sigma^j \models \phi_1 \tag{1.5.1f}$$

The formula $\dot{true}$ is modeled by every sequence, while $\dot{false}$ by no sequence. The formula ψ of the underlying logic L is evaluated in the first state of the sequence σ by using the satisfaction relation $\models_{\mathsf{L}}$. The semantics of the conjunction (Equation (1.5.1c)) and negation (Equation (1.5.1c)) are standard. The formula $\bigcirc \phi$ is modeled by a sequence $s_1 \cdot s_2 \cdot s_3 \ldots$ if and only if ϕ is modeled by the suffix $s_2 \cdot s_3 \ldots$. Finally, $\phi_1 \,\mathcal{U}\, \phi_2$ holds in σ if there exists a suffix σ^i of σ such that ϕ_2 is modeled by σ^i, and ϕ_1 holds in every suffix σ^j with $j < i$.

In this thesis we will sometimes omit parenthesis. To do so, we assume that $\dot{\neg}$ has the highest priority, while $\bigcirc$ has higher priority w.r.t. the remaining connectives and temporal operators.

LTL is suitable to model properties of concurrent and reactive systems. For instance, the formula $\Diamond\, \mathit{finish}$ is a reachability property that expresses that the state *finish* is eventually reached; the formula $\Box\, \dot{\neg}\, \mathit{error}$ is a safety property that states that an *error* never occurs. Other examples of specifications, such as fairness and mutual-exclusion properties, can be find in [97].

2 Timed Concurrent Constraint Programming

The concurrent constraint paradigm (*ccp* in short) is a simple but powerful model for concurrent systems. It is different from other programming paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model of von Neumann. In this way, the languages from this paradigm can easily deal with partial information: an underlying constraint system handles constraints on system variables. The formal definition of this programming paradigm can be found in [109, 110, 104].

In this chapter we describe the *Timed Concurrent Constraint Language* (*tccp* in short), introduced by [43], that adds to the original *ccp* model the notion of time –by defining a discrete and global clock[1]– and the ability to capture the absence of information. With these features, it is possible to specify behaviors typical of reactive systems such as *time-out* or *preemption*. A time-out waits for a limited period of time for an event to occur, if this event does not happen, then an exception is executed. A preemption consists in the ability of detecting an event and, as a consequence, aborting the current process and executing a new one.

In *tccp*, the computation progresses as the concurrent and asynchronous activity of several agents that can (monotonically) add (or *tell*) information in a *store*, and query (or *ask*) some information from that store. It is assumed that ask and tell actions take one time-unit to be executed. The parallel operator is interpreted in terms of *maximal parallelism* (in contrast to the interleaving approach of *ccp*), i.e., all the enabled agents of A and B are executed at the same time. The time response of the constraint solver is assumed to take a constant time, independently of the size of the store. In practice some restrictions (mentioned below) are taken in order to ensure that these hypothesis are reasonable (the reader can see [43] for details). In *tccp*, the absence of information is captured by a new operator (with respect to *ccp*):

$$\textsf{now}\ c\ \textsf{then}\ A\ \textsf{else}\ B$$

which tests if, in the current time instant, the store entails the constraint c and if it occurs, then in the same time instant it executes agent A; otherwise, it executes B (in the same time instant). It is necessary to fix a limit for the number of nested agents of this kind in order to ensure the bounded time response of the constraint solver. For recursive programs, such limit is ensured by the presence of the procedure call, since we assume that the evaluation of such a call takes one time unit.

[1]Differently from other languages where time is explicitly introduced by defining new *timing* agents.

$$
\begin{array}{rrll}
\text{(Programs)} & P ::= & D \,.\, A & \\
\\
\text{(Declarations)} & D ::= & p(\vec{x}) \mathrel{:-} A & \text{-definition} \\
 & \mid & D, D & \text{-conjunction} \\
\\
\text{(Agents)} & A ::= & \mathsf{skip} & \text{-skip} \\
 & \mid & \mathsf{tell}(c) & \text{-tell} \\
 & \mid & \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A & \text{-choice} \\
 & \mid & \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ A & \text{-conditional} \\
 & \mid & A \parallel A & \text{-parallel} \\
 & \mid & \exists x\, A & \text{-hiding} \\
 & \mid & p(x_1, \ldots, x_m) & \text{-procedure call}
\end{array}
$$

Figure 2.1: *tccp* syntax

2.1 Syntax

The *tccp* language is parametric to an underlying cylindric constraint system $\mathbf{C} = \langle \mathcal{C}, \leq, \otimes, \oplus, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$ (see Definition 1.4.1) such that $\langle \mathcal{C}, \leq, \otimes, \oplus, \mathit{true}, \mathit{false} \rangle$ is a complete lattice where $\otimes$ is the *lub* operator, $\oplus$ is the corresponding *glb* operator, and *true* and *false* are, respectively, the least and the greatest elements of $\mathcal{C}$. Moreover, *Var* is a denumerable set of variables with typical elements x, y, $z \ldots$ Finally, given $x \in \mathit{Var}$, $\exists_x : \mathcal{C} \to \mathcal{C}$ is the cylindrification (or hiding) operator.

Given a cylindric constraint system $\mathbf{C}$, the syntax of agents is given in Figure 2.1. We assume that c and c_i are finite constraints in $\mathcal{C}$, $p \in \Pi$, $x, x_1, \ldots, x_m \in \mathit{Var}$.

A *tccp* program P is an object of the form $D \,.\, A$, where A is an agent, called *initial agent*, and D is a set of *process declarations* of the form $p(\vec{x}) \mathrel{:-} A$ (for some agent A), where $\vec{x}$ denotes a generic tuple of variables.

The parallel and the hiding agents are inherited from the *ccp* model. The parallel agent represents the concurrency of the model in terms of *maximal parallelism*, while the hiding operator makes a variable local to some process. We can observe two additional agents which were present in *ccp*, but that here have a different semantics since they cause extension over time. The $\mathsf{tell}(c)$ agent adds the information c to the store, but this information is visible to other agents only in the following time instant. This means that the tell action takes one unit of time for its execution. The same thing occurs with the choice agent since the ask action takes also one unit of time to evaluate its guard. Also the procedure call consumes one time unit for its execution. Finally, the conditional agent $\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$ is the new agent introduced in the model in order to capture negative information. It behaves in a single instant of time in the sense that it evaluates the condition c and in the same instant of time it executes the corresponding agent. In particular, if the guard is satisfied, then A will be executed, otherwise the agent B will be executed. If we have two nested conditional agents, then the guards are recursively checked within the same time instant. This is the reason why we need a restriction about the maximum number of nested conditional agents.

2.2 Operational Semantics

In this section, we introduce the operational semantics of *tccp*. It is slightly different from the original one in [43] since we have introduced conditions in specific rules (namely Rules **R2**, **R4** and **R10**) in order to detect when the store becomes *false*. This modification follows the philosophy of computations defined for *ccp* in [110], where computations that reach an inconsistent store are considered failure computations. In [43], this check is not explicitly done. In our context, we are interested in detecting when a computation reaches *false*; however, once *false* is reached, no action can modify the store (*false* is the greatest element in the domain) and—after that moment—all guards in the program agents are always entailed, thus the computation from that instant has little interest. In particular we do not want to distinguish computations which end in *false* from those which loop on store *false*, contrarily to what [43] does.

It is worth noting that the modification of the rules alters the observables defined in [43] only by introducing some input-output pairs of computations that, at some point, reach the *false* store. This is due to the fact that [43] does not consider non-terminating computations and, with the new rules, a non-terminating computation that reaches the *false* store of [43] may result in a terminating computation with the new rules. For all the other cases of computations, the observables remain the same.

Definition 2.2.1 (Operational semantics of *tccp*) *The operational semantics of tccp is formally described by a transition system $T = (Conf, \rightarrow)$ where we assume that each transition step takes exactly one time-unit. Configurations in Conf are pairs $\langle A, c\rangle$ representing the agent to be executed (A) and the current global store (c). The transition relation $\rightarrow \subseteq Conf \times Conf$ is the least relation satisfying the rules **R1-R10** of Figure 2.2.*

As can be seen from the rules, the skip agent represents the successful termination of the computation. The $\mathsf{tell}(c)$ agent adds the constraint c to the current store and then stops. It takes one time-unit, thus the constraint c is visible to other agents from the following time instant. The store is updated by means of the $\otimes$ operator of the constraint system. The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ consults the store and non-deterministically executes (at the following time instant) one of the agents A_i whose corresponding guard c_i is entailed by the current store; otherwise, if no guard is entailed by the store, the agent suspends.

The conditional agent $\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$ behaves in the current time instant like A (respectively B) if c is (respectively is not) entailed by the store. Note that, because of the ability of *tccp* to handle partial information, $d \nvdash c$ is not equivalent to $d \vdash \neg c$. Thus, the else branch is taken not only when the condition is falsified, but also when there is not enough information to entail the condition. This characteristic is known in the literature as the ability to process "negative information" [106, 108]. $A \parallel B$ models the parallel composition of A and B in terms of maximal parallelism (in contrast to the interleaving approach of *ccp*), i.e., all the enabled agents of A and B are executed at the same time. The agent $\exists x\, A$ makes variable x local to A. To this end, it uses the $\exists$ operator of the constraint system. More specifically, it behaves like A with x considered local, i.e., the information on x provided by the external environment is hided to A, and the information on x produced by A is hided to the external world. In [43], an auxiliary construct $\exists^l x$ is used to explicitly show the store local to A. In particular, in Rule **R9**, the store l in

$$\frac{}{\langle \mathsf{tell}(c),\, d\rangle \rightarrow \langle \mathsf{skip},\, c \otimes d\rangle}\ d \neq \mathit{false} \tag{R1}$$

$$\frac{}{\langle \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i,\, d\rangle \rightarrow \langle A_j,\, d\rangle}\ j \in [1,n],\ d \vdash c_j,\ d \neq \mathit{false} \tag{R2}$$

$$\frac{\langle A,\, d\rangle \rightarrow \langle A',\, d'\rangle}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,\, d\rangle \rightarrow \langle A',\, d'\rangle}\ d \vdash c \tag{R3}$$

$$\frac{\langle A,\, d\rangle \not\rightarrow}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,\, d\rangle \rightarrow \langle A,\, d\rangle}\ d \vdash c,\ d \neq \mathit{false} \tag{R4}$$

$$\frac{\langle B,\, d\rangle \rightarrow \langle B',\, d'\rangle}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,\, d\rangle \rightarrow \langle B',\, d'\rangle}\ d \nvdash c \tag{R5}$$

$$\frac{\langle B,\, d\rangle \not\rightarrow}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,\, d\rangle \rightarrow \langle B,\, d\rangle}\ d \nvdash c \tag{R6}$$

$$\frac{\langle A,\, d\rangle \rightarrow \langle A',\, d'\rangle \quad \langle B,\, d\rangle \rightarrow \langle B',\, c'\rangle}{\langle A \parallel B,\, d\rangle \rightarrow \langle A' \parallel B',\, d' \otimes c'\rangle} \tag{R7}$$

$$\frac{\langle A,\, d\rangle \rightarrow \langle A',\, d'\rangle \quad \langle B,\, d\rangle \not\rightarrow}{\langle A \parallel B,\, d\rangle \rightarrow \langle A' \parallel B,\, d'\rangle} \qquad \frac{\langle A, d\rangle \not\rightarrow \quad \langle B, d\rangle \rightarrow \langle B', d'\rangle}{\langle A \parallel B, d\rangle \rightarrow \langle A \parallel B', d'\rangle} \tag{R8}$$

$$\frac{\langle A,\, l \otimes \exists_x d\rangle \rightarrow \langle B,\, l'\rangle}{\langle \exists^{l} x\, A,\, d\rangle \rightarrow \langle \exists^{l'} x\, B,\, d \otimes \exists_x l'\rangle} \tag{R9}$$

$$\frac{}{\langle p(\vec{x}),\, d\rangle \rightarrow \langle A,\, d\rangle}\ p(\vec{x}) \mathrel{:-} A \in D,\ d \neq \mathit{false} \tag{R10}$$

Figure 2.2: The transition system for *tccp*.

the agent $\exists^l x\, A$ represents the store local to A. This auxiliary operator is linked to the hiding construct by setting the initial local store to *true*, thus $\exists x\, A := \exists^{true} x\, A$. Finally, the agent $p(\vec{x})$ takes from D a declaration of the form $p(\vec{x})$:- A and then executes A at the following time instant. For the sake of simplicity, we assume that sets of declarations D are closed w.r.t. renaming of parameter names, i.e., if $p(\vec{x})$:- $A \in D$ then, for any $\vec{y} \in Var$, also $p(\vec{y})$:- $A\{\vec{x}/\vec{y}\} \in D$ [2].

2.3 Applications

Using the basic constructs presented in Figure 2.1 is possible to define other derived constructs, useful to model concurrent and reactive systems. For example, in [43] the *time-out* construct is introduced.

$$\sum_{i=1}^{i} \mathsf{ask}(c_i) \rightarrow A_i\ \mathsf{time\text{-}out}(m)\ B$$

This construct waits at most m time-units for the satisfaction of one of the guards c_i. Before this time limit, the process behaves like the choice construct, after waiting for m time units, if no guard is enabled, then this agent behaves as B.

Another additional primitive, presented in [43], is the *watchdog*:

$$\mathsf{do}\, A\, \mathsf{watching}\, c$$

This agent is the typical preemption primitive used to interrupt the activity of a process when some signal is presented. Namely it behaves as A as long c is not entailed by the store; when c is entailed the process A is immediately aborted. The reader can find more details about the semantics of those agents in [43].

It is possible to find in the literature different examples of systems that can be modelled using the *tccp* language.

The process declaration in Figure 2.3, presented in [53], models a subsystem of a microwave controller. The underlying constraint system is the Herbrand constraint system [41]. This process declaration detects if the door is open while the microwave is turned on. In that case, it forces that in the next time instant the microwave is turned-off and it emits an error signal (value 1); otherwise, the agent emits a signal of no error (value 0).

Due to the monotonicity of the store, streams are used to model *imperative-style* variables [43]. A stream S is a structure on the form $[v \mid T]$ where v is the instantiated value of the stream and T is a free variable representing the tail of the stream. The tail T can possibly be instantiated with a new value v', transforming S into a new stream on the form $[v,\ v' \mid T']$ where T' is the new tail of S and v' is the last instantiated value. The value of interest of a stream S is its last instantiated value which corresponds, roughly speaking, to the current value assigned to S.

In the example, the streams *Error*, *Door* and *Button* store the values that the simulated *modifiable variables* get along the computation. The first three tell agents link the future values of the streams with the *future streams* E, D and B. Then, when it is detected a

[2]This assumption is equivalent to use the diagonal elements of the constraint system: given the agent $p(\vec{x})$ and a declaration of the form $p(\vec{y})$:- A, we *diagonalize* the agent A before execution, i.e., we execute $\exists^{d_{x_1 y_1} \otimes \cdots \otimes d_{x_n y_n}} \vec{x}\, A$ at the following time instant.

$$
\begin{array}{l}
\mathit{microwave}(\mathit{Door}, \mathit{Button}, \mathit{Error}) \;\text{:-}\; \exists D\, \exists B\, \exists E\, \big(\\
\quad \mathsf{tell}(\mathit{Error} = [\_ \mid E]) \parallel \mathsf{tell}(\mathit{Door} = [\_ \mid D]) \parallel \mathsf{tell}(\mathit{Button} = [\_ \mid B]) \\
\quad \parallel \mathsf{now}(\mathit{Door} = [\mathit{open} \mid D] \wedge \mathit{Button} = [\mathit{on} \mid B]) \\
\qquad \mathsf{then}\; \big(\exists E1\, \mathsf{tell}(E = [1 \mid E1]) \parallel \exists B1\, \mathsf{tell}(B = [\mathit{off} \mid B1]) \big) \\
\qquad \mathsf{else}\; \exists E1\, \mathsf{tell}(E = [0 \mid E1]) \\
\quad \parallel \mathit{microwave}(D, B, E) \big)
\end{array}
$$

Figure 2.3: *tccp* microwave error controller

possible *risk* (characterized by the guard of the now agent), the microwave is turned off and an *error* signal is emitted (by the then branch of the conditional agent). The final recursive call restarts the same control at the next time instant. This check is made by using a conditional agent. If the door is opened when the microwave is turned-on, then the program forces that in the following time instant the microwave is turned-off and an error signal is emitted. If it is not true that the door is opened and the microwave is working, then the program simply emits a signal of no error in the following time instant.

Another clear example of reactive system is the one which models the railroad crossing problem. This is a very typical problem of critical reactive system which commonly appears in the literature (for example in [79, 43, 111]).

The system is composed by three main processes:

- *train* sends the message *near* to the controller when the train is approaching the crossing, and it send the message *out* when it has passed through the crossing.
- *controller* sends the order *down* to the gate each time it receives the signal *near* from a train; Similarly, when it receives the signal *out*, it sends the order *up* to the gate.
- *gate* changes its state to *down* when it receives the order *down* from the controller and to *up* if the order was *up*.

We show the *tccp* formalization of this system presented in [6]. Here, streams implement communication channels between processes.

The controller process (Figure 2.4) uses an *input channel C* through which it receives signals from the environment (trains), and an *output channel G* through which it sends orders to the gate process. It checks the input channel for a *near* signal (the guard in the first now agent), in which case it sends (tells) the order *down* through G, links the future values (C') of the stream C and restarts the check at the following time instant (recursive call $\mathit{controller}(C', G')$). If the *near* signal is not detected, then, the else branch looks for the *out* signal and (if present) behaves dually to the first branch. Finally, if no signal is detected at the current time instant (last else branch), then the process keeps checking from the following time instant (the process call takes one time instant).

The train process (Figure 2.5) notifies its state to the controller. Here $\mathsf{ask}(\mathit{true})^n$ denotes the n-times repetition of the agent $\mathsf{ask}(\mathit{true})$, and it corresponds to a delay of n

```
$controller(C,G)$ :- $\exists C', G'\big($
    now $(C = [near \mid \_])$ then
      tell$(C = [near \mid C'])$ ∥ tell$(G = [down \mid G'])$ ∥ $controller(C', G')$
    else  now $(C = [out \mid \_])$ then
            tell$(C = [out \mid C'])$ ∥ tell$(G = [up \mid G'])$ ∥ $controller(C', G')$
          else $controller(C, G)\big)$
```

Figure 2.4: *tccp* railroad crossing system controller

```
$train(C,T)$ :- $\exists C', C'', T', T''\big($
    ask$(true)$ → $train(C, T)$
    +
    ask$(true)$ → $\big($tell$(C = [near \mid C'])$ ∥
                        ask$(true)^{300}$ → $\big($tell$(T = [enter \mid T'])$ ∥
                                           ask$(true)^{20}$ → $($tell$(T' = [leave \mid T''])$ ∥
                                                              tell$(C' = [out \mid C''])$ ∥
                                                              $train(C'', T'')))\big)\big)$
```

Figure 2.5: *tccp* railroad crossing system train

time units. The process uses an *output channel* C to communicate with the controller and a *state stream* T. The process can simply recursively call itself (first branch of the ask agent) or either *non deterministically* send the *near* signal through C and after 300 time instants change its internal state T to *enter*. Then, after 20 time instants, it changes its state T to *leave*, it sends to the controller the *out* signal through the channel C and recursively calls itself.

The gate process (Figure 2.6) reacts to the signals from the controller. Orders are received through the input channel G and the state of the gate (represented by the stream S) is consequently updated. The ask agent (with two branches) makes the gate wait (suspend) until one of the guards is entailed, i.e., until one of the two orders is received. Once a signal is detected, after 100 time instants, the state of the gate is appropriately updated and a recursive call is done in order to keep the gate active (i.e., waiting for the successive order). Note that an instance of the gate process is run each time a signal is received, differently from the controller process which is run at each time instant.

Finally, the process *init* (Figure 2.7) models the whole railroad crossing system by composing in parallel *controller*, *train* and *gate*.

$$
\begin{aligned}
&gate(G, S) \mathrel{:\!-} \exists G', S' \big(\\
&\quad \mathsf{ask}(G = [\,down \mid \_\,]) \rightarrow \\
&\qquad \big(\mathsf{tell}(G = [\,down \mid G'\,]) \parallel \\
&\qquad\ \ \mathsf{ask}(true)^{100} \rightarrow (\mathsf{tell}(S = [\,down \mid S'\,]) \parallel gate(G', S'))\big) \\
&\quad + \\
&\quad \mathsf{ask}(G = [\,up \mid \_\,]) \rightarrow \\
&\qquad \big(\mathsf{tell}(G = [\,up \mid G'\,]) \parallel \\
&\qquad\ \ \mathsf{ask}(true)^{100} \rightarrow (\mathsf{tell}(S = [\,up \mid S'\,]) \parallel gate(G', S'))\big)\big)
\end{aligned}
$$

Figure 2.6: *tccp* railroad crossing system gate

$$
init \mathrel{:\!-} \exists C, T, S, G \big(train(C, T) \parallel controller(C, S) \parallel gate(S, G) \big)
$$

Figure 2.7: *tccp* railroad crossing system initialization

3 Small-step and Big-step Semantics

Abstract

In this chapter, we present a new compositional bottom-up semantics for *tccp* which is defined for the full language. In particular, is able to deal with the non-monotonic characteristic of the language, which constitutes a substantial additional technical difficulty w.r.t. other compositional denotational semantics present in literature (which do not tackle the full language).

The semantics is proven to be (correct and) fully abstract w.r.t. the full behavior of *tccp*, including infinite computations. This is particularly important since *tccp* has been defined to model reactive systems.

The overall of these features makes our proposal particularly suitable as the basis for the definition of semantic-based program manipulation tools (like analyzers, debuggers or verifiers), especially in the context of reactive systems.

Furthermore, we provide a big-step semantics (by abstraction of our small-step semantics) which tackles also outputs of infinite computations.

In the literature, much effort has been devoted to the development of *appropriate* denotational semantics for languages in the *ccp* paradigm (e.g. [47, 41, 49]). Compositionality and full abstraction are two highly desirable properties for a semantics, since they are needed for many purposes. A fully abstract model can be considered *the* semantics of a language [47].

In [41], the difficulties for handling nondeterminism and infinite behavior in the *ccp* paradigm were investigated. The authors showed that the presence of nondeterminism, local variables and synchronization require relatively complex structures for the denotational model of (non timed) *ccp* languages. In most *ccp* languages, nondeterminism is defined in terms of a global choice, which poses even more difficulties than a local-choice model [49].

Successively, [87] showed that for timed concurrent constraint languages, the presence of timing constructs which handle negative information in addition to non-determinism and local variables significantly complicates the definition of compositional and fully abstract semantics. Moreover, infinite behaviors (which become natural in the timed extensions) are an additional problem [41].

Presumably because of all these difficulties, for the languages of the *ccp* family which handle jointly the above mentioned features, the proposals of compositional semantics in the literature have been given by introducing (quite) severe restrictions on the languages. Essentially, they all limit the use of negative information and non-determinism, that are the distinguishing features that enhance the expressiveness of the paradigm w.r.t. other traditional ones. For us, this is contradictory and certainly unsatisfactory. Thus, we strived to develop a semantics which is fully abstract for the full *tccp* language. This is particularly important when one is interested in applying the semantics to develop (semantics-based) *fully automatic* program manipulation tools (like debuggers, verifiers and analyzers).

With this application in mind, we have developed a new (small-step) compositional, bottom-up, goal-independent and condensed semantics which is (correct and) *fully abstract* w.r.t. the small-step behavior of *full tccp*. To obtain this semantics the idea is to enrich the classical behavioral timed traces with information about the *essential* conditions that the store must (or must not) satisfy in order to make the program proceed with one or another execution branch. Thus, we associate conditions to the store of each computation step and then we collect just the most general hypothetical computations. Since conditions are constructed by using only the information in the guards of a program, we obtain a condensed semantics which also deals with non-monotonicity, because into denotations we have the *minimal* information needed to exploit computations arising from absence of information.

Since *tccp* was originally defined to model reactive systems, which many times include systems that do not terminate *with a purpose*, we have developed our semantics to distinguish among *terminating*, *suspending* and *non-terminating* computations. This improves the original semantics for *tccp* defined in [43] which identifies suspending and non-terminating computations. In particular, terminating computations are those that reach a point in which no agents are pending to be executed. Suspending computations are those that reach a point in which there are some agents pending to be executed, but there is not enough information in the store to entail the conditions that would make them evolve. We think it is essential to distinguish these two kinds of computations since, conceptually, a suspended computation has not completely finished its execution, and, in some cases, it could be a symptom of a system error.

To complete our proposal, we also define a big-step semantics (by abstraction of our small-step semantics) which tackles also outputs of infinite computations. We prove that its fragment for finite computations is (essentially) isomorphic to the traditional big-step semantics of [43]. Moreover, we also formally prove that it is not possible to have a correct input-output semantics which is defined *solely* on the information provided by the input/output pairs.

3.1 Small-step Semantics

In order to introduce the small-step semantics, we need first to define some (technical) notions. In the sequel, all definitions are parametric w.r.t. a cylindric constraint system $\mathbf{C} = \langle \mathcal{C}, \leq, \otimes, \oplus, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$. In the illustrative examples we will use, for the sake of simplicity, the Constraint System 1.4.3 of linear disequalities. We denote by $\mathbf{A}^{\Pi}_{\mathbf{C}}$ the set of agents and $\mathbf{D}^{\Pi}_{\mathbf{C}}$ the set of sets of process declarations built on signature Π and constraint system $\mathbf{C}$. By ϵ we denote the empty sequence; by $s_1 \cdot s_2$ the concatenation of two sequences s_1, s_2. We also abuse notation and, given a set of sequences S, by $s_1 \cdot S$ we denote $\{s_1 \cdot s_2 \,|\, s_2 \in S\}$.

Let us formalize first the notion of behavior of a set D of process declarations in terms of the transition system described in Figure 2.2. It collects all the small-step computations associated to D as the set of (all the prefixes of) the sequences of computation steps (in terms of sequences of stores), for all possible initial agents and stores.

Definition 3.1.1 *Let $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$. Then the* small-step (observable) behavior *of D is defined as:*

$$\mathcal{B}^{ss}[\![D]\!] := \bigcup_{\forall c \in \mathbf{C}, \forall A \in \mathbf{A}^{\Pi}_{\mathbf{C}}} \mathcal{B}^{ss}[\![D \,.\, A]\!]_c \qquad \textit{where}$$

$$\mathcal{B}^{ss}[\![D \,.\, A]\!]_{c_0} := \left\{ c_0 \cdot c_1 \cdot \ldots \cdot c_n \,\middle|\, \langle A, c_0 \rangle \to \langle A_1, c_1 \rangle \to \ldots \to \langle A_n, c_n \rangle \right\} \cup \{\epsilon\}$$

(where $\to$ is the transition relation given in Figure 2.2).

We call the sequences in $\mathcal{B}^{ss}[\![D \,.\, A]\!]_c$ behavioral timed traces *or simply* traces *(when clear from the context).*

We denote by $\approx_{ss}$ the equivalence relation between process declarations induced by $\mathcal{B}^{ss}$, namely for all $D_1, D_2 \in \mathbf{D}^{\Pi}_{\mathbf{C}}$, $D_1 \approx_{ss} D_2 \iff \mathcal{B}^{ss}[\![D_1]\!] = \mathcal{B}^{ss}[\![D_2]\!]$.

With this definition, we can formally state the requirement of full abstraction for semantics $\mathcal{S}$ as $\mathcal{S}[\![D_1]\!] = \mathcal{S}[\![D_2]\!] \iff D_1 \approx_{ss} D_2$.

To achieve a goal-independent semantics, a typical solution is to define denotations by using only the most general traces (in our case those for the weakest store) *plus* define a suitable semantic operator which can reconstruct the semantics of any expression (in our case agent) from such most general denotations. This result can be achieved in this way only if the set of all traces for each expression is itself *condensing* (borrowing the terminology from program analysis [74, 80]), which in our case means that the set of all traces for an agent A with initial store c can be reconstructed from the set of all traces of A with initial store *true*. The problem in following this approach in the *tccp* case is that $\mathcal{B}^{ss}$ is not condensing, since not all behavioral timed traces can be retrieved from the most general ones. This is due to the ask, now and hiding constructs. For instance, consider the agent $A :=$ now $x = 3$ then tell$(z = 0)$ else tell$(z = 1)$. Given the initial store *true*, we obtain the trace $true \cdot z = 1$, while for the stronger initial store $x = 3$ we obtain the trace $x = 3 \cdot (x = 3 \wedge z = 0)$, which is not comparable to the former (since $z = 0 \not\Rightarrow z = 1$ and $z = 1 \not\Rightarrow z = 0$). Hence, the latter trace cannot be obtained from the former trace, which has been generated for the *most general* store. Indeed—in general—in *tccp*, given $S := \mathcal{B}^{ss}[\![D \,.\, A]\!]_c$ (the set of traces for an agent A with initial store c), if we compute $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d$ with a stronger initial store d ($d \vdash c$), then some traces of S may disappear and, what is more critical, new traces, *which are not instances* of the ones in S, can appear. In the community of the *ccp* paradigm [41, 108], this characteristic is known as "non-monotonicity of the language".

Because of *tccp*'s non-monotonicity, $\mathcal{B}^{ss}$ is also not compositional. For instance, consider the agents $A_1 :=$ tell$(x = 1)$ and

$$A_2 := \mathsf{ask}(true) \to \mathsf{now}\ (x = 1)\ \mathsf{then}\ \mathsf{tell}(y = 0)\ \mathsf{else}\ \mathsf{tell}(y = 1)$$

For each c, $\mathcal{B}^{ss}[\![\varnothing \,.\, A_1]\!]_c = \{c \cdot (x = 1 \wedge c)\}$. Moreover, for each c that implies[1] $x = 1$, $\mathcal{B}^{ss}[\![\varnothing \,.\, A_2]\!]_c = \{c \cdot c \cdot (y = 0 \wedge c)\}$ while, when $c \not\Rightarrow (x = 1)$, $\mathcal{B}^{ss}[\![\varnothing \,.\, A_2]\!]_c = \{c \cdot c \cdot (y = 1 \wedge c)\}$. Now, for the parallel composition of these agents $A_1 \parallel A_2$, $\mathcal{B}^{ss}[\![\varnothing . A_1 \parallel A_2]\!]_{true} = \{true \cdot (x = 1) \cdot (x = 1 \wedge y = 0)\}$ which cannot be computed by *merging* the traces of A_1 and A_2.

[1]We recall that in the exemplification cylindric constraint system, the entailment is logical implication.

Thus, it does not come as a surprise that for the majority of non-monotonic languages of the *ccp* paradigm, the compositional semantics that have been written [41, 42, 49, 88, 51, 90, 50] are not defined for the full language, either because they avoid the constructs that cause non-monotonicity or because they restrict their use. Hence, the ability to handle non-monotonicity (and thus the full language without any limitation) in a condensed way is certainly one of the strengths of this thesis.

The example above shows why, due to the non-monotonicity of *tccp*, in order to obtain a compositional (and goal-independent) semantics *for the full language* it is not possible to follow the traditional strategy and collect in the semantics the traces associated to the weakest initial store. Actually, we have found the solution to the problem of compositionality by trying to solve another (related) problem. Since in a top-down (goal-dependent) approach the (initial) current store is propagated, then the decisions regarding a conditional or choice agent (where the computation evolves depending on the entailment of the guards in the current store) can be taken immediately. However, if we want to define a fixpoint semantics which builds the denotations bottom-up we have the problem that, while we are building the fixpoint, we do not know the current store yet. Thus, it is impossible to know *which* execution branch *has to be* taken in correspondence of a program's guard.

To solve both problems our proposal is to enrich behavioral timed traces with information about the essential conditions that the store must (or must not) satisfy in order to make the program proceed with one or another execution branch. Thus, we associate conditions to the store of each computation step and then we collect (only) the most general hypothetical computations. These conditions are constructed by using the information in the guards of the ask and now constructs of a program.

We will formally show that this indeed solves both the problem of constructing bottom-up the semantics and of having a compositional and condensed semantics coping with non-monotonicity.

3.1.1 The semantic domain

Let us start by introducing the notion of condition, that is the base to build our denotations. Intuitively, we need "positive conditions" for branches related to the entailment of guards and "negative conditions" for non-entailment, i.e., for the branches where the current store does not entail the associated condition.

Definition 3.1.2 (Conditions) *A* condition η*, over Cylindric Constraint System* $\mathbf{C}$*, is a pair* $\eta = (\eta^+, \eta^-)$ *where*

- $\eta^+ \in \mathbf{C}$ *is called* positive condition*, and*
- $\eta^- \in \wp(\mathbf{C})$ *is called* negative condition.

A condition is valid *when* $\eta^+ \neq \mathit{false}$*,* $\mathit{true} \notin \eta^-$ *and* $\forall c \in \eta^-.\, \eta^+ \nvdash c$*. We denote* $\Lambda_{\mathbf{C}}$ *the set of all conditions and* $\Delta_{\mathbf{C}}$ *the subset of valid ones.*

The conjunction of two conditions $\eta_1 = (\eta_1^+, \eta_1^-)$ *and* $\eta_2 = (\eta_2^+, \eta_2^-)$ *is defined (by abuse of notation) as* $\eta_1 \otimes \eta_2 := (\eta_1^+ \otimes \eta_2^+, \eta_1^- \cup \eta_2^-)$*. Two conditions are called* incompatible *if their conjunction is not valid.*

A store $c \in \mathbf{C}$ *is* consistent *with* η*, written* $c \gg \eta$*, if* $\eta^+ \otimes c \neq \mathit{false}$ *and* $\forall h \in \eta^-.\, c \nvdash h$*. Moreover, we say that* c satisfies η*, written* $c \Vdash \eta$*, when* $c \vdash \eta^+$ *and* $\forall h \in \eta^-.\, c \nvdash h$*.*

We extend the $\exists_x$ *operator to conditions as* $\exists_x(\eta^+, \eta^-) := (\exists_x \eta^+, \exists_x \eta^-)$*.*

Due to the partial nature of the constraint system, for negative conditions we cannot use the *glb* (disjunction) $\bigoplus_{i=1}^{n} c_i$ instead of set $\{c_1, \dots, c_n\}$ since we can have a store c such that $c \vdash \bigoplus_{i=1}^{n} c_i$ while $\forall i. c \nvdash c_i$. For instance, we can have two guards $x > 2$ and $x \leq 2$ and it may happen that the current store does not satisfy any of them, but their *glb* $x > 2 \oplus x \leq 2$ (which is *true*) is entailed by any store.

Clearly, if a store—different from *false*—satisfies a condition, then it is also consistent with that condition. If two conditions are incompatible, then there exists no constraint $c \in \mathbf{C} \setminus \{\mathit{false}\}$ that entails simultaneously both conditions.

Now we are ready to enrich with conditions the notion of trace.

Definition 3.1.3 (Conditional state) *A* conditional state*, over Cylindric Constraint System* $\mathbf{C}$*, is one of the following constructs.*

Conditional store. *A pair* $\eta \rightarrowtail c$*, for each* $\eta \in \Lambda_{\mathbf{C}}$ *and* $c \in \mathbf{C}$*.*

Stuttering. *The construct* $\mathit{stutt}(C)$*, for each finite* $C \subseteq \mathbf{C} \setminus \{\mathit{true}\}$*.*

End-of-process. *The construct* $\boxtimes$*.*

In a conditional store $t = \eta \rightarrowtail c$*, the constraint* c *is the* store *of* t*. We say that* $\eta \rightarrowtail c$ *is* valid *if* η *is valid. We extend* $\exists_x$ *to conditional states as* $\exists_x \big((\eta^+, \eta^-) \rightarrowtail c\big) := \exists_x(\eta^+, \eta^-) \rightarrowtail \exists_x c$*,* $\exists_x \mathit{stutt}(C) := \mathit{stutt}(\exists_x C)$ *and* $\exists_x \boxtimes := \boxtimes$*.*

The conditional store $\eta \rightarrowtail c$ is used to represent a hypothetical computation step where η is the condition that the current store must satisfy in order to make the computation proceed. Moreover, c represents the information that is added to the global store in the next time instant in case η is satisfied.

The stuttering $\mathit{stutt}(C)$ is needed to model the suspension of the computation due to an ask construct, i.e., it represents the fact that there is no guard in C (the guards of a choice agent) entailed by the current store.

Definition 3.1.4 (Conditional trace) *A* conditional trace *(over Cylindric Constraint System* $\mathbf{C}$*) is a (possibly infinite) sequence* $t_1 \cdots t_n \cdots$ *of valid conditional states (over* $\mathbf{C}$*)—where* $\boxtimes$ *can be used only as a terminator—that respects the following properties:*

Monotonicity. *For each* $t_i = \eta_i \rightarrowtail c_i$ *and* $t_j = \eta_j \rightarrowtail c_j$ *such that* $j \geq i$*,* $c_j \vdash c_i$*.*
Consistency. *For each* $t_i = \eta_i \rightarrowtail c_i$ *and* t_{i+1} *either of the form* $(\eta_{i+1}^+, \eta_{i+1}^-) \rightarrowtail c_{i+1}$ *or* $\mathit{stutt}(\eta_{i+1}^-)$*, we have that* $\forall c^- \in \eta_{i+1}^-. c_i \nvdash c^-$*.*

We denote by $\mathbf{CT}_{\mathbf{C}}$ *the set of all conditional traces, or simply write* $\mathbf{CT}$ *when clear from the context.*

The sequence of stores of a given conditional trace s *is the sequence of stores* c_j *of all conditional states* $t_j = \eta_j \rightarrowtail c_j$ *of* s*. The* limit store *of a (finite or infinite) trace* s *is the lub of the stores (of the conditional states) of* s*.*

A finite conditional trace that is ended with $\boxtimes$ *as well as an infinite conditional trace is said* failed *or* (finitely) successful *depending on whether its limit store* c *is false or not respectively. Such* c *is called* computed result*.*

A sequence (of conditional states) that does not satisfy these properties is called an invalid trace*.*

Each conditional trace models a hypothetical *tccp* computation: for each time instant, we have a conditional state where each condition represents the information that the global store has to satisfy in order to proceed to the next time instant.

The Monotonicity property is needed since in *tccp*, as well as in *ccp* but not in all its extensions, each store in a computation entails the previous ones. Note that because of this, for any *finite* conditional trace $t_1, \dots, t_n$ whose sequence of stores (of the conditional stores) is $c_1, \dots, c_m$ ($m \le n$), the limit store $\otimes_{i=1}^{m} c_i$ is just the last store c_m.

The Consistency property affirms that the store of a given conditional state cannot be in contradiction with the condition associated to the successive conditional state.

Example 3.1.5

It is easy to verify that the sequence $r_1 \coloneqq (\mathit{true}, \varnothing) \rightarrowtail y = 0 \cdot (x > 2, \varnothing) \rightarrowtail y = 0 \wedge z = 3 \cdot \boxtimes$ is a conditional trace. The first component of the trace states that in the first time instant the store $y = 0$ is computed in any case (the condition $(\mathit{true}, \varnothing)$ is always satisfied). The second component requires the constraint $x > 2$ to be satisfied by the (global) store in order to proceed by adding to the next state the information $z = 3$. Instead, the sequence $r_2 \coloneqq (\mathit{true}, \varnothing) \rightarrowtail x = 0 \cdot (x = 0, \varnothing) \rightarrowtail \mathit{true} \cdot \boxtimes$ is not a conditional trace since the Monotonicity property does not hold because $\mathit{true} \nvdash x = 0$. Also $r_3 \coloneqq (\mathit{true}, \varnothing) \rightarrowtail x = 0 \cdot \mathit{stutt}(\{x \ge 0\}) \cdot \boxtimes$ is not a conditional trace: it does not satisfy the Consistency property since $x = 0$ implies the (only) negative condition in the successive conditional state ($x \ge 0$).

Note that finite conditional traces not ending in $\boxtimes$ are partial traces that can still evolve and thus they are always a prefix of a longer conditional trace.

Definition 3.1.6 (Semantic domain) *A set $R \subseteq \mathbf{CT}$ is* closed by prefix *if for each $r \in R$, all the prefixes p of r (denoted as $p \le_{\mathit{pref}} r$) are also in R.*

We denote the domain of non-empty *sets of conditional traces that are closed by prefix as $\mathbf{P}$ (i.e., $\mathbf{P} \coloneqq \{R \subseteq \mathbf{CT} \mid R \neq \varnothing, r \in R \Rightarrow \forall p \le_{\mathit{pref}} r. p \in R\}$).*

We order elements in $\mathbf{P}$ by set inclusion $\subseteq$.

It is worth noting that $(\mathbf{P}, \subseteq, \bigcup, \bigcap, \mathbf{CT}, \{\epsilon\})$ is a complete lattice.

This conceptual representation is pretty simple, especially to understand the lattice structure, considered the fact that we admit infinite traces. However, each prefix-closed set contains a lot of redundant traces, which are quite inconvenient for technical definitions. Thus, we will use an equivalent representation obtained by considering the crown of prefix-closed sets. Namely, given $P \in \mathbf{P}$, we remove all the prefixes of a trace in the set with the function $\mathit{maximal}(P) \coloneqq \{r \in P \mid \nexists p \in P \setminus \{r\}. r \le_{\mathit{pref}} p\}$. Let $\mathbf{M} \coloneqq \mathit{maximal}(\mathbf{CT})$, $\mathbf{M} \coloneqq \{\mathit{maximal}(P) \mid P \in \mathbf{P}\}$ and call *maximal conditional trace sets* the elements of $\mathbf{M}$. The inverse of map *maximal* is, for each $M \in \mathbf{M}$,

$$\mathit{prefix}(M) \coloneqq \left\{p \in \mathbf{CT} \,\middle|\, p \le_{\mathit{pref}} r, r \in M\right\} \tag{3.1.1}$$

The order of $\mathbf{M}$ is induced from the one in $\mathbf{P}$ as $M_1 \sqsubseteq M_2 \iff \mathit{prefix}(M_1) \subseteq \mathit{prefix}(M_2)$ which is equivalent to say that $M_1 \sqsubseteq M_2 \iff \forall r_1 \in M_1 \exists r_2 \in M_2. r_1 \le_{\mathit{pref}} r_2$. We define the *lub* $\bigsqcup$ and the *glb* $\bigsqcap$ of $\mathbf{M}$ analogously. It is straightforward to prove that $(\mathbf{P}, \subseteq) \underset{\mathit{maximal}}{\overset{\mathit{prefix}}{\leftrightarrows}} (\mathbf{M}, \sqsubseteq)$ is an *order-preserving isomorphism*, so $(\mathbf{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}, \{\epsilon\})$ is also a complete lattice.

Although this second representation is very convenient for technical definitions, it is not very suited for examples. For instance, different maximal traces have frequently (significant) common prefixes; hence, some parts have to be written many times and, more important, it can be difficult to visualize the repetition (obfuscating the comprehension). Thus, in our examples we will use another equivalent representation in terms of prefix trees. Namely, we will use trees with (non root) nodes labeled with conditional states. Given $P \in \mathbf{P}$, $\mathit{tree}(P)$ builds the prefix tree of P, obtained by combining all the sequences that have a prefix in common in the same path. Let $\mathbf{T} := \{\mathit{tree}(P) \mid P \in \mathbf{P}\}$. The inverse of *tree* is the function $\mathit{path}\colon \mathbf{T} \to \mathbf{P}$ which returns the set of all possible paths starting from the root. Let $\trianglelefteq$ be the order on $\mathbf{T}$ induced by the order on $\mathbf{P}$, i.e., $T_1 \trianglelefteq T_2 \iff \mathit{path}(T_1) \subseteq \mathit{path}(T_2)$. We define the *lub* and *glb* of $\mathbf{T}$ in a similar way. It is straightforward to prove that $(\mathbf{P}, \subseteq) \underset{\mathit{tree}}{\overset{\mathit{path}}{\leftrightarrows}} (\mathbf{T}, \trianglelefteq)$ is an order-preserving isomorphism, so also $(\mathbf{T}, \trianglelefteq)$ is a complete lattice. Finally, by function composition we can define a third order-preserving isomorphism $(\mathbf{M}, \sqsubseteq) \underset{\mathit{prefix} \circ \mathit{tree}}{\overset{\mathit{path} \circ \mathit{maximal}}{\leftrightarrows}} (\mathbf{T}, \trianglelefteq)$ between trees and maximal conditional traces. In the sequel we will use the representation which is most convenient in each case.

3.1.2 Fixpoint denotations of programs

The technical core of our semantics definition is the agent semantics evaluation function (Definition 3.1.16, page 34) which, given an agent A and an interpretation $\mathcal{I}$ (for the process symbols of A), builds the maximal conditional traces associated to A. To define it, we need first to introduce some auxiliary semantic functions.

Definition 3.1.7 (Propagation Operator) *Let $r \in \mathbf{M}$ and $c \in \mathbf{C}$. We define the* propagation *of c in r, written $r{\downarrow_c}$, by structural induction as $\boxtimes{\downarrow_c} = \boxtimes$, $\epsilon{\downarrow_c} = \epsilon$ and*

$$((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\downarrow_c} = \begin{cases} (\eta^+ \otimes c, \eta^-) \rightarrowtail d \otimes c \cdot (r'{\downarrow_c}) & \text{if } c \gg (\eta^+, \eta^-),\ d \otimes c \neq \mathit{false} \\ (\eta^+ \otimes c, \eta^-) \rightarrowtail \mathit{false} \cdot \boxtimes & \text{if } c \gg (\eta^+, \eta^-),\ d \otimes c = \mathit{false} \end{cases}$$

$$(\mathit{stutt}(\eta^-) \cdot r'){\downarrow_c} = \mathit{stutt}(\eta^-) \cdot (r'{\downarrow_c}) \qquad \text{if } \forall c^- \in \eta^- .\, c \not\vdash c^-$$

We abuse notation and denote by $R{\downarrow_c}$ the point-wise extension of ${\downarrow_c}$ to sets of conditional traces: $R{\downarrow_c} := \{r{\downarrow_c} \mid r \in R$ and $r{\downarrow_c}$ is defined$\}$.

This operator is used in the definition of the semantics of constructs that add new information to traces. By definition, the *propagation operator* $\downarrow$ is a partial function $\mathbf{M} \times \mathbf{C} \to \mathbf{M}$ that instantiates a conditional trace with a given constraint and checks the consistency of the new information with the conditional states in the trace. This information needs to be propagated also to the successive (i.e., future) conditional states in order to maintain the monotonicity of the store.

Example 3.1.8

Given the conditional trace $r := (\mathit{true}, \varnothing) \rightarrowtail x > 10 \cdot (\mathit{true}, \varnothing) \rightarrowtail x > 20 \cdot \boxtimes$, the propagation of $y > 2$ in r ($r{\downarrow_{y>2}}$) is $(y > 2, \varnothing) \rightarrowtail x > 10 \wedge y > 2 \cdot (y > 2, \varnothing) \rightarrowtail x > 20 \wedge y > 2 \cdot \boxtimes$.

For $r' := (\mathit{true}, \{y > 0\}) \rightarrowtail \mathit{true} \cdot \boxtimes$ the propagation $r'{\downarrow_{y>2}}$ is not defined since $y > 2 \not\gg (\mathit{true}, \{y > 0\})$.

Finally, given the conditional trace $r'' := (\mathit{true}, \varnothing) \rightarrowtail y < 0 \cdot \boxtimes$, the propagation $r''{\downarrow_{y>2}}$ produces the conditional trace $(y > 2, \varnothing) \rightarrowtail \mathit{false} \cdot \boxtimes$ since $y > 2 \gg (\mathit{true}, \varnothing)$ and $y < 0 \wedge y > 2 = \mathit{false}$.

Note that the consecutive propagation of two constraints $(r{\downarrow_c}){\downarrow_{c'}}$ is equivalent to $r{\downarrow_{(c \otimes c')}}$ (as stated formally in Lemma 3.A.2).

Definition 3.1.9 (c-compatible) *$r \in \mathbf{M}$ is said to be* compatible *w.r.t. $c \in \mathbf{C}$ (c-compatible in short) if, for each $(\eta^+, \eta^-) \rightarrowtail d$ in r, $c \gg (\eta^+, \eta^-)$, and for each $\mathit{stutt}(\eta^-)$ in r, $c \nvdash c^-$ for all $c^- \in \eta^-$.*

When r is not c-compatible w.r.t. c, the store c is in contradiction with a condition of some conditional state of r and then $r{\downarrow_c}$ is not defined.

The following parallel composition auxiliary operator is used in the definition of the semantics of the parallel construct. Intuitively, this operator combines (with maximal parallelism) the information coming from two conditional traces and it checks the satisfiability of the conditions and the consistency of the resulting stores.

Definition 3.1.10 (Parallel composition) *The* parallel composition *partial operator $\bar{\|} : \mathbf{M} \times \mathbf{M} \to \mathbf{M}$ is the commutative closure of the following partial operation defined by structural induction as: $r \mathbin{\bar{\|}} \epsilon := r$, $r \mathbin{\bar{\|}} \boxtimes := r$ and*

$$(\mathit{stutt}(\eta_1^-) \cdot r_1') \mathbin{\bar{\|}} (\mathit{stutt}(\eta_2^-) \cdot r_2') := \mathit{stutt}(\eta_1^- \cup \eta_2^-) \cdot (r_1' \mathbin{\bar{\|}} r_2')$$

Moreover, if $\eta_1 \otimes \eta_2$ is valid, r_1' is c_2-compatible and r_2' is c_1-compatible, then

$$(\eta_1 \rightarrowtail c_1 \cdot r_1') \mathbin{\bar{\|}} (\eta_2 \rightarrowtail c_2 \cdot r_2') := \begin{cases} \eta_1 \otimes \eta_2 \rightarrowtail c_1 \otimes c_2 \cdot ((r_1'{\downarrow_{c_2}}) \mathbin{\bar{\|}} (r_2'{\downarrow_{c_1}})) & \text{if } c_1 \otimes c_2 \neq \mathit{false} \\ \eta_1 \otimes \eta_2 \rightarrowtail \mathit{false} \cdot \boxtimes & \text{if } c_1 \otimes c_2 = \mathit{false}, \end{cases}$$

Finally, if $\forall c^- \in \eta_2^- . \eta_1^+ \nvdash c^-$ and r_2' is c_1-compatible, then

$$((\eta_1^+, \eta_1^-) \rightarrowtail c_1 \cdot r_1') \mathbin{\bar{\|}} (\mathit{stutt}(\eta_2^-) \cdot r_2') := (\eta_1^+, \eta_1^- \cup \eta_2^-) \rightarrowtail c_1 \cdot (r_1' \mathbin{\bar{\|}} (r_2'{\downarrow_{c_1}}))$$

Clearly, by definition, $\bar{\|}$ is commutative. Moreover, because of $\otimes$ associativity, $\bar{\|}$ is also associative. It is worth noting that, if one of the traces is not compatible with the propagated constraint, then the parallel composition is not defined.

Example 3.1.11

Consider $r_1 := (\mathit{true}, \varnothing) \rightarrowtail y > 2 \cdot (y > 2, \varnothing) \rightarrowtail y > 2 \cdot \boxtimes$ and $r_2 := (z = 1, \varnothing) \rightarrowtail z = 1 \cdot \boxtimes$. Since r_1 and r_2 do not share variables, the compatibility checks always succeed and then $r_1 \mathbin{\bar{\|}} r_2 = (z = 1, \varnothing) \rightarrowtail y > 2 \wedge z = 1 \cdot (y > 2 \wedge z = 1, \varnothing) \rightarrowtail y > 2 \wedge z = 1 \cdot \boxtimes$.

Consider now $r_3 := \mathit{stutt}(\{y > 0\}) \cdot (y > 0, \varnothing) \rightarrowtail y > 0 \wedge z = 3 \cdot \boxtimes$. Traces r_1 and r_3 share the variable y and it can be seen that the information regarding y in the two traces is consistent, thus $r_1 \mathbin{\bar{\|}} r_3 = (\mathit{true}, \{y > 0\}) \rightarrowtail y > 2 \cdot (y > 2, \varnothing) \rightarrowtail y > 2 \wedge z = 3 \cdot \boxtimes$.

Finally, consider $r_4 := (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdot (\mathit{true}, \{y > 0\}) \rightarrowtail \mathit{true} \cdot \boxtimes$. This trace, in the second time instant, requires that the constraint $y > 0$ cannot be entailed by the current store. However, the trace r_1 states, at the same time instant, that $y > 2$. This is the reason because $r_1 \mathbin{\bar{\|}} r_4$ is not defined.

Note that $\downarrow$ distributes over $\bar{\|}$, in the sense that $(r_1 \mathbin{\bar{\|}} r_2){\downarrow_c} = (r_1{\downarrow_c}) \mathbin{\bar{\|}} (r_2{\downarrow_c})$ (as stated formally in Lemma 3.A.3).

The last auxiliary operator that we need is the hiding operator $\bar{\exists}\colon \mathit{Var} \times \mathbf{M} \to \mathbf{M}$ which, intuitively, hides the information regarding a given variable in a conditional trace.

Definition 3.1.12 (Hiding operator) *Given $r \in \mathbf{M}$ and $x \in \mathcal{V}$, we define the* hiding of x in r, *written $\bar{\exists}_x r$, by structural induction as $\bar{\exists}_x \epsilon := \epsilon$, $\bar{\exists}_x \boxtimes := \boxtimes$,*

$$\bar{\exists}_x \big((\eta^+, \eta^-) \rightarrowtail c \cdot r'\big) := \exists_x \big((\eta^+, \eta^-) \rightarrowtail c\big) \cdot \bar{\exists}_x r'$$
$$\bar{\exists}_x \big(\mathit{stutt}(\eta^-) \cdot r'\big) := \exists_x \mathit{stutt}(\eta^-) \cdot \bar{\exists}_x r'$$

We distinguish two special classes of conditional traces.

Definition 3.1.13 (Self-sufficient and x-self-sufficient conditional trace) *A maximal trace $r \in \mathbf{M}$ is said to be* self-sufficient *if the first condition is $(\mathit{true}, \varnothing)$ and, for each $t_i = \eta_i \rightarrowtail c_i$ and $t_{i+1} = \eta_{i+1} \rightarrowtail c_{i+1}$, $c_i \Vdash \eta_{i+1}$ (each store satisfies the successive condition).*

Moreover, r is self-sufficient w.r.t. $x \in \mathcal{V}$ *(*x-self-sufficient*) if $\bar{\exists}_{\mathit{Var} \setminus \{x\}} r$ is self-sufficient.*

Definition 3.1.13 is stronger than Definition 3.1.4 since the latter does not require satisfiability but just consistency of the store w.r.t. conditions. Informally, this new definition demands that for self-sufficient conditional traces, no additional information (from other agents) is needed in order to *complete* the computation. In an x-self-sufficient conditional trace the same happens but only considering information about variable x.

Example 3.1.14

The conditional trace r_1 of Example 3.1.5 is not self-sufficient since $y = 0 \not\Vdash x > 2$.

Now consider a variation where we add the information $x = 4$ to the stores, namely $r_2 := (\mathit{true}, \varnothing) \rightarrowtail y = 0 \wedge x = 4 \cdot (x > 2, \varnothing) \rightarrowtail y = 0 \wedge z = 3 \wedge x = 4 \cdot \boxtimes$. It is easy to see that r_2 is a self-sufficient conditional trace, essentially because we add enough information in the first store to satisfy the second condition, i.e., $y = 0 \wedge x = 4 \Vdash (x > 2, \varnothing)$.

Moreover, r_2 is also x-self-sufficient since $\bar{\exists}_{\mathit{Var} \setminus \{x\}} r_2 = (\mathit{true}, \varnothing) \rightarrowtail x = 4 \cdot (x > 2, \varnothing) \rightarrowtail x = 4 \cdot \boxtimes$, which is a self-sufficient trace.

Interpretations

Now we introduce the notion of interpretation, which is used to give meaning to process calls by associating to each process symbol a set of (maximal) conditional traces "modulo variance".

Definition 3.1.15 (Interpretations) *Let $\mathbf{PC}_\Pi := \{p(\vec{x}) \mid p \in \Pi, \vec{x}$ are distinct variables$\}$ (or simply $\mathbf{PC}$ when clear from the context).*

Two functions $I, J\colon \mathbf{PC} \to \mathbf{M}$ are variants*, denoted by $I \cong J$, if for each $\pi \in \mathbf{PC}$ there exists a variable renaming ρ such that $(I(\pi))\rho = J(\pi\rho)$.*

An interpretation *is a function $\mathcal{I}\colon \mathbf{PC} \to \mathbf{M}$ modulo variance*[2].

The semantic domain $\mathbf{I}_\Pi$ (or simply $\mathbf{I}$ when clear from the context) is the set of all interpretations ordered by the pointwise extension of $\sqsubseteq$ (which by an abuse of notation we also denote by $\sqsubseteq$).

[2]i.e., a family of elements of $\mathbf{M}$ indexed by $\mathbf{PC}$ modulo variance.

The partial order on $\mathbf{I}$ formalizes the evolution of the computation process. $(\mathbf{I}, \sqsubseteq)$ is a complete lattice and its least upper bound and greatest lower bound are the pointwise extension of $\bigsqcup$ and $\bigsqcap$, respectively. In the sequel we abuse the notations of $\mathbf{M}$ for $\mathbf{I}$ as well. The bottom element is $\bot_{\mathbf{I}} := \lambda\pi.\,\{\epsilon\}$.

Essentially, we define the semantics of each predicate in Π over formal parameters whose names are actually irrelevant. It is important to note that $\mathbf{PC}_\Pi$ (modulo variance) has the same cardinality of Π (and is thus finite) and therefore each interpretation is a finite collection of (possibly infinite) elements. Hence, in the sequel, we explicitly write interpretations by cases, like

$$\mathcal{I} := \begin{cases} \pi_1 \mapsto T_1 \\ \vdots \\ \pi_n \mapsto T_n \end{cases} \qquad \text{representing} \qquad \begin{array}{c} \mathcal{I}(\pi_1) := T_1 \\ \vdots \\ \mathcal{I}(\pi_n) := T_n \end{array}$$

In the following, any $\mathcal{I} \in \mathbf{I}$ is implicitly considered as an arbitrary function $\mathbf{PC} \to \mathbf{M}$ obtained by choosing an arbitrary representative of the elements of $\mathcal{I}$ generated by $\cong$. Actually, all the operators that we use on $\mathbf{I}_\Pi$ are also independent of the choice of the representative. Therefore, we can define any operator on $\mathbf{I}$ in terms of its counterpart defined on functions $\mathbf{PC} \to \mathbf{M}$.

Moreover, we also implicitly assume that the application of an interpretation $\mathcal{I}$ to a process call π, denoted by $\mathcal{I}(\pi)$, is the application $I(\pi)$ of any representative I of $\mathcal{I}$ which is defined exactly on π. For example, if $\mathcal{I} = (\lambda p(x,y).\,\{(true,\varnothing) \rightarrowtail x = y\})/_{\cong}$ then $\mathcal{I}(p(u,v)) = \{(true,\varnothing) \rightarrowtail u = v\}$.

Semantics Evaluation Function of Agents

We are finally ready to define the evaluation function of an agent A w.r.t. an interpretation $\mathcal{I}$, which computes the set of (maximal) conditional traces associated to the agent A. It is important to note that the computation does not depend on an initial store. Instead, the weakest (most general) condition for each agent is (computed and) accumulated in the conditional traces.

Definition 3.1.16 (Semantics Evaluation Function for Agents) *Given $A \in \mathbf{A}^\Pi_\mathbf{C}$ and $\mathcal{I} \in \mathbf{I}_\Pi$, we define the* semantics evaluation $\mathcal{A}[\![A]\!]_\mathcal{I} \in \mathbf{M}$ *by structural induction as follows.*

$$\mathcal{A}[\![\mathsf{skip}]\!]_\mathcal{I} := \{\boxtimes\} \tag{3.1.2}$$

$$\mathcal{A}[\![\mathsf{tell}(c)]\!]_\mathcal{I} := \{(true,\varnothing) \rightarrowtail c \cdot \boxtimes\} \tag{3.1.3}$$

$$\mathcal{A}[\![A \parallel B]\!]_\mathcal{I} := \bigsqcup\{r_A \mathbin{\bar{\parallel}} r_B \mid r_A \in \mathcal{A}[\![A]\!]_\mathcal{I},\, r_B \in \mathcal{A}[\![B]\!]_\mathcal{I}\} \tag{3.1.4}$$

$$\mathcal{A}[\![\exists x\, A]\!]_\mathcal{I} := \bigsqcup\{\bar{\exists}_x\, r \mid r \in \mathcal{A}[\![A]\!]_\mathcal{I},\, r \textit{ is } x\textit{-self-sufficient}\} \tag{3.1.5}$$

$$\mathcal{A}[\![p(\vec{x})]\!]_\mathcal{I} := (true,\varnothing) \rightarrowtail true \cdot \mathcal{I}(p(\vec{x}))^3 \tag{3.1.6}$$

$$\mathcal{A}[\![\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_\mathcal{I} := lfp_\mathbf{M}\, \lambda R.\Big(stutt(\{c_1,\ldots,c_n\}) \cdot R \sqcup \bigsqcup\big\{(c_i,\varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}}) \,\big|\, 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_\mathcal{I},\, r\ c_i\textit{-compatible}\big\}\Big) \tag{3.1.7}$$

^{3}Recall that by $s_1 \cdot S$ we denote $\{s_1 \cdot s_2 \,|\, s_2 \in S\}$.

$$\begin{aligned}
&\mathcal{A}[\![\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\mathcal{I}} :=\\
&\quad \{(c,\varnothing) \rightarrowtail c\cdot\boxtimes \mid \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{I}}\} \sqcup && (3.1.8a)\\
&\quad \bigsqcup\{(\eta^+ \otimes c, \eta^-) \rightarrowtail d\otimes c\cdot(r{\downarrow_c}) \mid (\eta^+,\eta^-) \rightarrowtail d\cdot r \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\\
&\qquad\qquad d\otimes c \neq \mathit{false},\ \forall c^- \in \eta^-.\, \eta^+\otimes c \nVdash c^-,\ r\ \textit{c-compatible}\} \sqcup && (3.1.8b)\\
&\quad \bigsqcup\{(\eta^+ \otimes c, \eta^-) \rightarrowtail \mathit{false}\cdot\boxtimes \mid (\eta^+,\eta^-) \rightarrowtail d\cdot r \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\\
&\qquad\qquad d\otimes c = \mathit{false},\ \forall c^- \in \eta^-.\, \eta^+\otimes c \nVdash c^-,\ r\ \textit{c-compatible}\ \} \sqcup && (3.1.8c)\\
&\quad \bigsqcup\{(c,\eta^-) \rightarrowtail c\cdot(r{\downarrow_c}) \mid \mathit{stutt}(\eta^-)\cdot r \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\ \forall c^- \in \eta^-.\, c \nVdash c^-,\ r\ \textit{c-compatible}\} \sqcup\\
&&& (3.1.8d)\\
&\quad \bigsqcup\{(\mathit{true},\{c\}) \rightarrowtail \mathit{true}\cdot\boxtimes \mid \boxtimes \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} \sqcup && (3.1.8e)\\
&\quad \bigsqcup\{(\eta^+,\eta^- \cup \{c\}) \rightarrowtail d\cdot r \mid (\eta^+,\eta^-) \rightarrowtail d\cdot r \in \mathcal{A}[\![B]\!]_{\mathcal{I}},\ \eta^+ \nVdash c\} \sqcup && (3.1.8f)\\
&\quad \bigsqcup\{(\mathit{true},\eta^- \cup \{c\}) \rightarrowtail \mathit{true}\cdot r \mid \mathit{stutt}(\eta^-)\cdot r \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} && (3.1.8g)
\end{aligned}$$

By $\mathit{lfp}(F)$ we denote the least fixed point of any monotonic function $F\colon \mathcal{L} \to \mathcal{L}$, over some lattice $\mathcal{L}$.

We now explain in detail each case of the definition.

(3.1.2) The semantics of the skip agent contains just the trace composed of the end-of-process construct that marks the end of the computation.

(3.1.3) For the tell(c) agent we have a trace with two conditional states, the first one with condition $(\mathit{true},\varnothing)$ since c must be added to the store in any case (in the next time instant). Next, the computation terminates with the end-of-process symbol $\boxtimes$.

(3.1.4) The semantics for the parallel composition of two agents is defined in terms of the auxiliary operator $\bar{\|}$, explained in Definition 3.1.10.

(3.1.5) The hiding construct must hide the information about x from all traces that cannot be altered by the presence of external information about x, thus the hiding operation is applied just to x-self-sufficient conditional traces (Definition 3.1.13), that are those for which no additional information about variable x is needed (from other agents) in order to complete the computation.

(3.1.6) The semantics of process call $p(\vec{x})$ simply delays by one time instant the traces for $p(\vec{x})$ in interpretation $\mathcal{I}$ by prefixing them with $(\mathit{true},\varnothing) \rightarrowtail \mathit{true}$.

(3.1.7) The semantics for the non-deterministic choice collects, for each guard c_i, a conditional trace of the form $(c_i,\varnothing) \rightarrowtail c_i\cdot(r{\downarrow_{c_i}})$. This trace requires that c_i has to be satisfied by the current store (positive part of the condition in the first state). Then, the constraint c_i is propagated to the trace r (the continuation of the computation, which belongs to the semantics of A_i). Note that the requirement of c_i-compatibility ensures that $r{\downarrow_{c_i}}$ is defined.

Furthermore, we collect the stuttering traces, which correspond to the case when the computation suspends. These traces are of the form $\mathit{stutt}(\{c_1,\dots,c_n\})\cdot r$ where r is, recursively, an element of the semantics of the choice agent.

(3.1.8) The definition for the conditional agent now c then A else B is similar to the previous case. However, since the now construct must be instantaneous, in order to correctly model the timing of the agent we have seven cases depending on the possible forms of the first conditional state of the semantics of A (respectively B), on the value of the resulting store (*false* or not) and on the fact that the guard c is satisfied or not in the current time instant.

(3.1.8a)–(3.1.8d) represent the case in which the guard c is satisfied by the current store. In this case, the agent now must behave instantaneously as A. For this reason, we distinguish four different cases corresponding to the possible form of conditional traces associated to A. In particular, (3.1.8a) corresponds to the case when the computation of A ends, thus also the computation of the conditional agent must end. In (3.1.8b), the information added (in one step) by A is compatible with the condition and with the rest of the computation and, moreover, does not produce *false* when merged—by using $\otimes$—with the current store d. (3.1.8c) stops the conditional trace since the information produced by A added to the current store produces the inconsistent store *false*. Finally, (3.1.8d) corresponds to the case when A suspends.

(3.1.8e)–(3.1.8g) model the cases when c is not entailed by the current store. In this situation, the agent now must behave instantaneously as B, and the definition follows the same reasoning as for (3.1.8a), (3.1.8b) and (3.1.8d). The main difference is that, instead of adding c to the positive condition in the first conditional state, we add $\{c\}$ to the negative condition.

In the sequel, we use a standard notation for the iterates of the computation of the least fixpoint of a monotonic function $F\colon \mathcal{L} \to \mathcal{L}$, over lattice $\mathcal{L}$ whose bottom is $\bot$ and *lub* is $\bigsqcup$. Namely, $F{\uparrow}k$ denotes, for each $k \in \mathbf{N}$, $F^k(\bot)$ and $F{\uparrow}\omega$ denotes $\bigsqcup\{F^k(\bot) \mid k \in \mathbf{N}\}$. Recall that, for a continuos F, $\mathit{lfp}(F) = F{\uparrow}\omega$.

Example 3.1.17

Let us evaluate the semantics for the *tccp* agent $A_1 := A_2 \parallel A_3$ where

$$\begin{aligned}
A_2 &:= \mathsf{tell}(y = 2) \parallel \mathsf{tell}(x = y)\\
A_3 &:= \mathsf{ask}(\mathit{true}) \to \mathsf{now}\ (x = 0)\ \mathsf{then}\ \mathsf{tell}(z > 0)\ \mathsf{else}\ A_4\\
A_4 &:= \mathsf{ask}(y \geq 0) \to \mathsf{tell}(z \leq 0)
\end{aligned}$$

Since there are no process calls, the interpretation $\mathcal{I}$ is irrelevant for the result. We start by computing the semantics for A_4, i.e., $\mathcal{A}[\![A_4]\!]_{\mathcal{I}} = \mathit{lfp}_{\mathbf{M}}(F)$ where

$$\begin{aligned}
F(R) &:= \{r\} \sqcup \mathit{stutt}(\{y \geq 0\}) \cdot R \quad \text{and}\\
&\quad r := (y \geq 0, \varnothing) \rightarrowtail y \geq 0 \cdot (y \geq 0, \varnothing) \rightarrowtail y \geq 0 \wedge z \leq 0 \cdot \boxtimes
\end{aligned}$$

The iterates of F are:

$$\begin{aligned}
F{\uparrow}1 &= F(\{\epsilon\}) = \{r,\ \mathit{stutt}(\{y \geq 0\})\}\\
F{\uparrow}2 &= F(F{\uparrow}1) = \{r,\ \mathit{stutt}(\{y \geq 0\}) \cdot r,\ \mathit{stutt}(\{y \geq 0\}) \cdot \mathit{stutt}(\{y \geq 0\})\}\\
&\ \ \vdots\\
\mathit{lfp}_{\mathbf{M}}(F) &= \Big\{\big(\mathit{stutt}(\{y \geq 0\})\big)^n \cdot r \,\Big|\, n \in \mathbf{N}\Big\} \sqcup \{\mathit{stutt}(\{y \geq 0\}) \cdots \mathit{stutt}(\{y \geq 0\}) \cdots\}
\end{aligned}$$

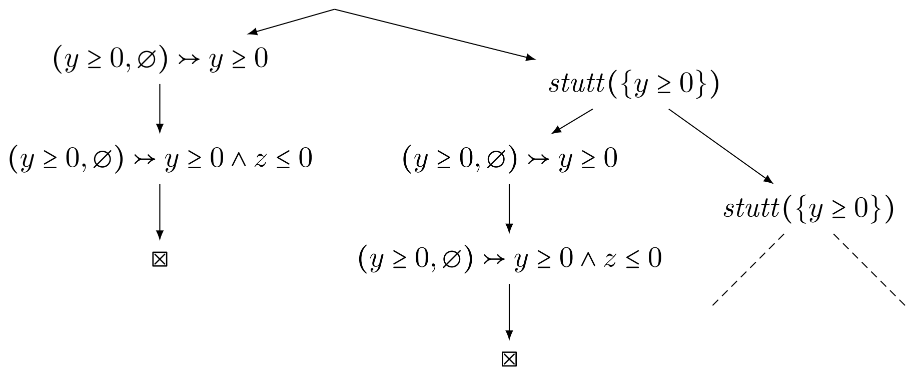

$(y \geq 0, \varnothing) \rightarrowtail y \geq 0$
$stutt(\{y \geq 0\})$
$(y \geq 0, \varnothing) \rightarrowtail y \geq 0 \wedge z \leq 0$
$(y \geq 0, \varnothing) \rightarrowtail y \geq 0$
$stutt(\{y \geq 0\})$
$\boxtimes$
$(y \geq 0, \varnothing) \rightarrowtail y \geq 0 \wedge z \leq 0$
$\boxtimes$

Figure 3.1: Tree representation of $\mathcal{A}[\![A_4]\!]_{\mathcal{I}}$ in Example 3.1.17.

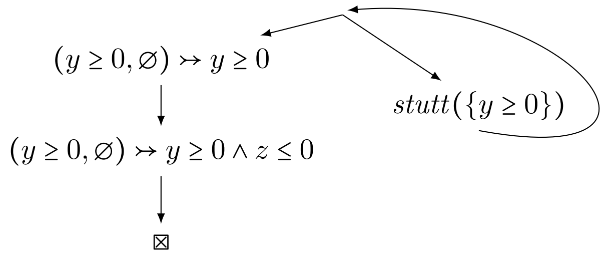

$(y \geq 0, \varnothing) \rightarrowtail y \geq 0$
$stutt(\{y \geq 0\})$
$(y \geq 0, \varnothing) \rightarrowtail y \geq 0 \wedge z \leq 0$
$\boxtimes$

Figure 3.2: Graph representation of $\mathcal{A}[\![A_4]\!]_{\mathcal{I}}$ in Example 3.1.17.

Figure 3.1 graphically represents $\mathcal{A}[\![A_4]\!]_{\mathcal{I}}$, which consists of a trace for the case in which the guard is satisfied, and a set of traces for the case in which it suspends. As it can be observed, the tree in Figure 3.1 consists of an infinite replication of the same pattern. We can depict such infinite trees as finite graphs, as in Figure 3.2. The back-loop arc is just a graphical shortcut which represents the (infinite) tree that is obtained by unrolling the loop. It is important to note that nodes reached by a path of length 2 (via the back-loop arc) have to be considered as a single arc, thus corresponding just to a one time instant delay.

With the semantics of A_4, we compute $\mathcal{A}[\![A_3]\!]_{\mathcal{I}} = \{r_1, r_2\} \cup R$ where

$$\begin{aligned}
r_1 &:= (true, \varnothing) \rightarrowtail true \cdot (x = 0, \varnothing) \rightarrowtail x = 0 \wedge z > 0 \cdot \boxtimes \\
r_2 &:= (true, \varnothing) \rightarrowtail true \cdot (y \geq 0, \{x = 0\}) \rightarrowtail y \geq 0 \cdot (y \geq 0, \varnothing) \rightarrowtail y \geq 0 \wedge z \leq 0 \cdot \boxtimes \\
R &:= (true, \varnothing) \rightarrowtail true \cdot (true, \{y \geq 0,\, x = 0\}) \rightarrowtail true \cdot \mathcal{A}[\![A_4]\!]_{\mathcal{I}}
\end{aligned}$$

All the traces of $\mathcal{A}[\![A_3]\!]_{\mathcal{I}}$ start with the conditional store $(true, \varnothing) \rightarrowtail true$ corresponding to the ask agent with guard *true*. The trace r_1 corresponds to the case when (in the current time instant) the guard $x = 0$ is satisfied; the trace r_2 corresponds to $x = 0$ not satisfied and $y \geq 0$ satisfied; while we have R when none is satisfied and A_4 is executed.

Now we can compute the semantics for A_1 by parallel composition of $\mathcal{A}[\![A_3]\!]_{\mathcal{I}}$ with $\mathcal{A}[\![A_2]\!]_{\mathcal{I}} = \{(true, \varnothing) \rightarrowtail (y = 2 \wedge x = y) \cdot \boxtimes\}$.

The combination of the trace r_1 in $\mathcal{A}[\![A_3]\!]_{\mathcal{I}}$ with the trace in $\mathcal{A}[\![A_2]\!]_{\mathcal{I}}$ does not produce contributes since the constraint $y = 2$, when propagated to the second component of r_1, is in contradiction with the positive part of the condition ($y = 2 \wedge x = y \wedge x = 0 \equiv false$).

Indeed, $(\mathit{true},\varnothing) \rightarrowtail (y = 2 \wedge x = y) \cdot ((x = 0,\varnothing) \rightarrowtail x = 0 \wedge z > 0 \cdot \boxtimes){\downarrow}_{(y=2\wedge x=y)} = (\mathit{true}, \varnothing) \rightarrowtail (y = 2 \wedge x = y) \cdot (\mathit{false},\varnothing) \rightarrowtail \mathit{false} \cdot \boxtimes$ is not a trace since $(\mathit{false},\varnothing)$ is not a valid condition.

The combination of the set of traces R (corresponding to the suspension of the agent A_4) and the $\mathsf{tell}(y = 2)$ agent also produces no trace. Definition 3.1.16 prescribes to compute $(\mathit{true},\varnothing) \rightarrowtail y = 2 \wedge x = y \cdot \bigsqcup\{((\mathit{true},\{y \geq 0,\ x = 0\}) \rightarrowtail \mathit{true} \cdot r'){\downarrow}_{(y=2\wedge x=y)} \mid r' \in \mathcal{A}[\![A_4]\!]_{\mathcal{I}}\}$, which is empty, since $y = 2 \wedge x = y \not\gg (\mathit{true},\{y \geq 0,\ x = 0\})$ because $y = 2 \wedge x = y \Rightarrow y \geq 0$. These traces would correspond to the suspension of the agent A_4, and this can happen only when $y \geq 0$ is not satisfied, but the first component of the parallel agent tells $y = 2$ (thus $y \geq 0$ is satisfied). Therefore, only the combination of the trace r_2 in $\mathcal{A}[\![A_3]\!]_{\mathcal{I}}$ and the trace of $\mathcal{A}[\![A_2]\!]_{\mathcal{I}}$ produces a trace. Namely

$$\begin{aligned}\mathcal{A}[\![A_1]\!]_{\mathcal{I}} = \{&(\mathit{true},\varnothing) \rightarrowtail (y = 2 \wedge x = y) \cdot (y = 2 \wedge x = y, \{x = 0\}) \rightarrowtail (y = 2 \wedge x = y) \cdot \\ &(y = 2 \wedge x = y,\varnothing) \rightarrowtail (y = 2 \wedge x = y \wedge z \leq 0) \cdot \boxtimes\}\end{aligned}$$

Due to the partial nature of the constraint system, the combination of the hiding operator with non-determinism can make the language behavior non-monotonic. As already mentioned, this is the reason because for all the languages of the *ccp* paradigm, the compositional semantics that have been written either avoid non-monotonic and/or non-deterministic constructs or restrict their use. Let us show now that we are able to handle the following example, which is an adaptation to *tccp* of the one used in [42, 88] to illustrate the non-monotonicity problem.

Example 3.1.18

Consider the non-monotonic agent

$$A := \mathsf{ask}(x = 1) \rightarrow \mathsf{tell}(\mathit{true}) + \mathsf{ask}(\mathit{true}) \rightarrow \mathsf{tell}(y = 2).$$

It is easy to see that for the initial store *true* just the second branch can be taken, whereas for the (greater) initial store $x = 1$, the two branches can be executed.

Since there are no process calls, for any interpretation $\mathcal{I}$, $\mathcal{A}[\![A]\!]_{\mathcal{I}} = \{r_1,\ r_2\}$, where

$$\begin{aligned}r_1 &:= (x = 1,\varnothing) \rightarrowtail x = 1 \cdot (x = 1,\varnothing) \rightarrowtail x = 1 \cdot \boxtimes\\ r_2 &:= (\mathit{true},\varnothing) \rightarrowtail \mathit{true} \cdot (\mathit{true},\varnothing) \rightarrowtail y = 2 \cdot \boxtimes\end{aligned}$$

We have two possible traces depending on whether the initial store is strong enough to entail $x = 1$ or not.

[42, 88] show that within their semantics they do not collect all possible evaluations for agent $A' := \mathsf{tell}(x = 1) \parallel \exists x\, A$. On the contrary, in our case, since

$$\begin{aligned}\bar{\exists}_{\mathit{Var}\smallsetminus\{x\}}\, r_1 &= (x = 1,\varnothing) \rightarrowtail x = 1 \cdot (x = 1,\varnothing) \rightarrowtail x = 1 \cdot \boxtimes\\ \bar{\exists}_{\mathit{Var}\smallsetminus\{x\}}\, r_2 &= (\mathit{true},\varnothing) \rightarrowtail \mathit{true} \cdot (\mathit{true},\varnothing) \rightarrowtail \mathit{true} \cdot \boxtimes\end{aligned}$$

only r_2 is x-self-sufficient and, by Definition 3.1.16,

$$\mathcal{A}[\![\exists x\, A]\!]_{\mathcal{I}} = \{(\mathit{true},\varnothing) \rightarrowtail \mathit{true} \cdot (\mathit{true},\varnothing) \rightarrowtail y = 2 \cdot \boxtimes\}.$$

By composing we have

$$\mathcal{A}[\![A']\!]_{\mathcal{I}} = \{(\mathit{true},\varnothing) \rightarrowtail x = 1 \cdot (x = 1,\varnothing) \rightarrowtail y = 2 \wedge x = 1 \cdot \boxtimes\}.$$

It is easy to see that the information on the variable x added by the tell agent does not affect the *internal* execution of the agent A, as expected.

There are some technical decisions that ensure the correctness of the defined semantics. One can note that in the definition of the propagation operator (Definition 3.1.7), the propagated information is added not only to the store of the state, but also to the (positive part of the) condition. This means that the positive part of the conditions in a trace contains not only the information that has to be satisfied up to that computation step, but also the constraints that have been added during computation in the previous time instants. From the computations in the examples above, it may seem that the propagation of the accumulated information in the conditions of the states could be redundant. However, it is necessary in order to have full abstraction w.r.t. the behavior, otherwise we would distinguish agents whose behavior is actually the same, as shown in the following example.

Example 3.1.19

Consider the following two (very similar) agents:

$$A_1 := \mathsf{ask}(x > 2) \rightarrow \mathsf{tell}(y = 1) \qquad\qquad A_2 := \mathsf{ask}(x > 4) \rightarrow \mathsf{tell}(y = 1)$$

We have similar but different semantics. Namely,

$$\begin{aligned}
\mathcal{A}[\![A_1]\!]_{\mathcal{I}} &= \{\big(\,\mathit{stutt}(\{x > 2\})\big)^n \cdot r_1 \,|\, n \in \mathbf{N}\} \sqcup \{\mathit{stutt}(\{x > 2\}) \cdots \mathit{stutt}(\{x > 2\}) \cdots\} \\
r_1 &= (x > 2, \varnothing) \rightarrowtail \mathit{true} \cdot (x > 2, \varnothing) \rightarrowtail y = 1 \cdot \boxtimes \\
\mathcal{A}[\![A_2]\!]_{\mathcal{I}} &= \{\big(\,\mathit{stutt}(\{x > 4\})\big)^n \cdot r_2 \,|\, n \in \mathbf{N}\} \sqcup \{\mathit{stutt}(\{x > 4\}) \cdots \mathit{stutt}(\{x > 4\}) \cdots\} \\
r_2 &= (x > 4, \varnothing) \rightarrowtail \mathit{true} \cdot (x > 4, \varnothing) \rightarrowtail y = 1 \cdot \boxtimes
\end{aligned}$$

However, consider now the following two agents, which embed A_1 and A_2 in the same context:

$$A_1' := \mathsf{tell}(x = 7) \parallel \mathsf{ask}(\mathit{true}) \rightarrow A_1 \qquad\qquad A_2' := \mathsf{tell}(x = 7) \parallel \mathsf{ask}(\mathit{true}) \rightarrow A_2$$

Then, the two traces corresponding to the satisfaction of the guards are, respectively:

$$r_3 = (\mathit{true}, \varnothing) \rightarrowtail x = 7 \cdot r_1{\downarrow_{(x=7)}} \qquad\qquad r_4 = (\mathit{true}, \varnothing) \rightarrowtail x = 7 \cdot r_2{\downarrow_{(x=7)}}$$

Since the propagated constraint is stronger than the guards in both the agents, the resulting compositions are the same. In fact, thanks to the accumulation of the store in the condition, we do not distinguish them:

$$r_1{\downarrow_{(x=7)}} = r_2{\downarrow_{(x=7)}} = (\mathit{true}, \varnothing) \rightarrowtail x = 7 \cdot (x = 7, \varnothing) \rightarrowtail x = 7 \cdot (x = 7, \varnothing) \rightarrowtail x = 7 \wedge y = 1 \cdot \boxtimes$$

This is correct since A_1' and A_2' have the same behavior. On the contrary, if the constraint $x = 7$ were not added to the condition, but only to the store of the state, then we would have two different conditional traces for these two agents. Thus, we would lose the full abstraction, since we would distinguish two agents that behave in the same way.

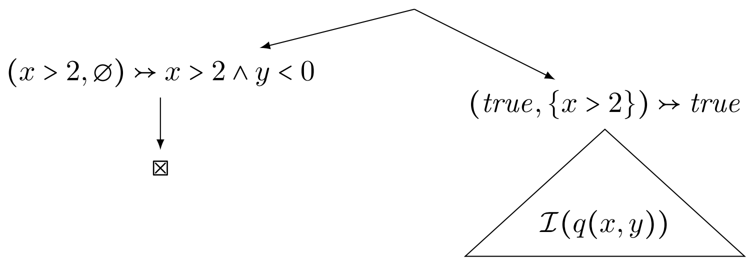

$(x > 2, \varnothing) \rightarrowtail x > 2 \wedge y < 0$
$\boxtimes$
$(true, \{x > 2\}) \rightarrowtail true$
$\mathcal{I}(q(x,y))$

Figure 3.3: Tree representation for $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ in Example 3.1.21.

Fixpoint Denotations of Process Declarations

Now we can finally define the semantics for a set of process declarations D.

Definition 3.1.20 (Fixpoint semantics) *Given $D \in \mathbf{D}_{\mathbf{C}}^{\Pi}$, we define $\mathcal{D}[\![D]\!]{:}\,\mathbf{I} \to \mathbf{I}$, for each $p \in \Pi$, as*

$$\mathcal{D}[\![D]\!]_{\mathcal{I}}(p(\vec{x})) := \bigsqcup\left\{\mathcal{A}[\![A]\!]_{\mathcal{I}} \,\middle|\, p(\vec{x}) \coloneq A \in D\right\}.$$

The fixpoint denotation *of D is $\mathcal{F}[\![D]\!] := lfp(\mathcal{D}[\![D]\!]) = \mathcal{D}[\![D]\!]{\uparrow}\omega$.*

We denote with $\approx_{\mathcal{F}}$ the equivalence relation on $\mathbf{D}_{\mathbf{C}}^{\Pi}$ induced by $\mathcal{F}$. Namely, $D_1 \approx_{\mathcal{F}} D_2 \iff \mathcal{F}[\![D_1]\!] = \mathcal{F}[\![D_2]\!]$.

The semantics of a tccp program $D\,.\,A$ is $\mathcal{P}[\![D\,.\,A]\!] := \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$.

$\mathcal{F}[\![D]\!]$ is well defined since $\mathcal{D}[\![D]\!]$ is continuous (as stated formally in Lemma 3.A.5).

Let us show how the semantics for a set of process declarations is computed by means of some examples.

Example 3.1.21

Let $D := \{q(x,y) \coloneq A\}$ where

$$A := \mathsf{now}\ (x > 2)\ \mathsf{then}\ \mathsf{tell}(y < 0)\ \mathsf{else}\ q(x,y).$$

Intuitively, the agent waits until x is greater than 2. Once the global store is strong enough to entail this condition, the constraint $y < 0$ is added to the store and the computation ends.

First we need to compute, for each $\mathcal{I} \in \mathbf{I}$, the evaluation of the body of the process declaration. Namely,

$$\mathcal{A}[\![A]\!]_{\mathcal{I}} = \{\bar{r}\} \sqcup \{(true, \{x > 2\}) \rightarrowtail true \cdot s \mid s \in \mathcal{I}(q(x,y))\}$$

where $\bar{r} := (x > 2, \varnothing) \rightarrowtail x > 2 \wedge y < 0 \cdot \boxtimes$. Intuitively, the trace $\bar{r}$ corresponds to the then branch of the conditional agent, whereas the else branch is represented by a set of traces, one for each trace in the interpretation of the process call. $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ is graphically represented in Figure 3.3.

The iterates of $\mathcal{D}[\![D]\!]$ are

$$\mathcal{D}[\![D]\!]{\uparrow}1 = \Big\{q(x,y) \mapsto \{\bar{r},\ (true, \{x > 2\}) \rightarrowtail true\}$$

$$\mathcal{D}[\![D]\!]{\uparrow}2 = \begin{cases} q(x,y) \mapsto & \{\bar{r},\ (true, \{x > 2\}) \rightarrowtail true \cdot \bar{r}, \\ & (true, \{x > 2\}) \rightarrowtail true \cdot (true, \{x > 2\}) \rightarrowtail true\} \end{cases}$$

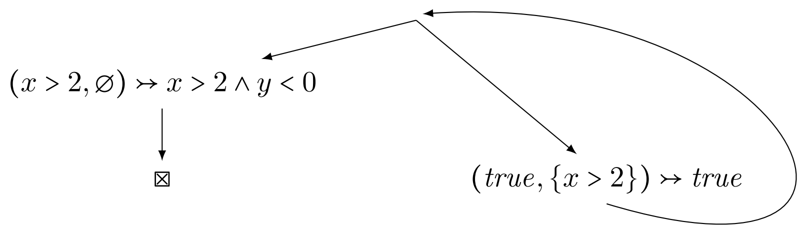

$(x > 2, \varnothing) \rightarrowtail x > 2 \wedge y < 0$
$\boxtimes$
$(\mathit{true}, \{x > 2\}) \rightarrowtail \mathit{true}$

Figure 3.4: Graph representation of the fixpoint $\mathcal{F}[\![D]\!](q(x,y))$ in Example 3.1.21.

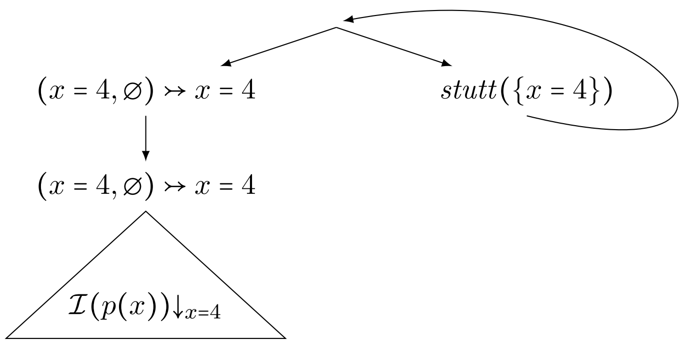

$(x = 4, \varnothing) \rightarrowtail x = 4$
$\mathit{stutt}(\{x = 4\})$
$(x = 4, \varnothing) \rightarrowtail x = 4$
$\mathcal{I}(p(x))\downarrow_{x=4}$

Figure 3.5: Graph representation for $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ in Example 3.1.22.

$$\vdots$$

$$\mathcal{D}[\![D]\!]\uparrow\omega = \begin{cases} q(x,y) \mapsto & \{\big((\mathit{true}, \{x > 2\}) \rightarrowtail \mathit{true}\big)^n \cdot \bar{r} \,\big|\, n \in \mathbf{N}\} \\ & \sqcup\{(\mathit{true}, \{x > 2\}) \rightarrowtail \mathit{true} \cdots (\mathit{true}, \{x > 2\}) \rightarrowtail \mathit{true} \cdots\} \end{cases}$$

The limit $\mathcal{F}[\![D]\!](q(x,y)) = (\mathcal{D}[\![D]\!]\uparrow\omega)(q(x,y))$ is graphically represented in Figure 3.4.

Example 3.1.22

Let $D := \{p(x) \mathrel{:-} A\}$ where $A := \mathsf{ask}(x = 4) \to p(x)$. First we need to compute, for each $\mathcal{I} \in \mathbf{I}$, the evaluation of the body of the process declaration. Namely,

$$\begin{aligned} \mathcal{A}[\![A]\!]_{\mathcal{I}} = & \{\big(\mathit{stutt}(\{x = 4\})\big)^n \cdot \bar{r} \cdot s \,\big|\, n \in \mathbf{N},\, s \in \mathcal{I}(p(x))\} \sqcup \\ & \{\mathit{stutt}(\{x = 4\}) \cdots \mathit{stutt}(\{x = 4\}) \cdots\} \end{aligned}$$

where $\bar{r} := (x = 4, \varnothing) \rightarrowtail x = 4 \cdot (x = 4, \varnothing) \rightarrowtail x = 4$. It is worth noticing that the second conditional state of $\bar{r}$ corresponds to the delay that is introduced each time that a process call is run. $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ is graphically represented in Figure 3.5.

The iterates of $\mathcal{D}[\![D]\!]$ are

$$\mathcal{D}[\![D]\!]\uparrow 1 = \begin{cases} p(x) \mapsto & \{(\mathit{stutt}(\{x = 4\}))^n \cdot \bar{r} \,|\, n \in \mathbf{N}\} \sqcup \\ & \{\mathit{stutt}(\{x = 4\}) \cdots \mathit{stutt}(\{x = 4\}) \cdots\} \end{cases}$$

$$\mathcal{D}[\![D]\!]\uparrow 2 = \begin{cases} p(x) \mapsto & \{(\mathit{stutt}(\{x = 4\}))^n \cdot \bar{r} \cdot \bar{r} \,|\, n \in \mathbf{N}\} \sqcup \\ & \{\mathit{stutt}(\{x = 4\}) \cdots \mathit{stutt}(\{x = 4\}) \cdots\} \end{cases}$$

$$\vdots$$

$$\mathcal{F}[\![D]\!] = \begin{cases} p(x) \mapsto & \{(\mathit{stutt}(\{x = 4\}))^n \cdot \bar{r} \cdots \bar{r} \cdots \,|\, n \in \mathbf{N}\} \sqcup \\ & \{\mathit{stutt}(\{x = 4\}) \cdots \mathit{stutt}(\{x = 4\}) \cdots\} \end{cases}$$

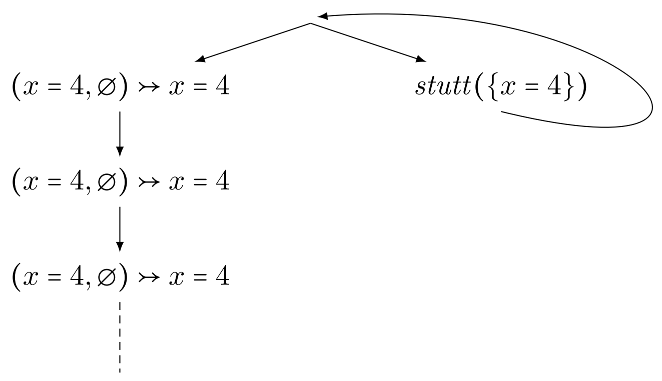

$(x = 4, \varnothing) \rightarrowtail x = 4$
$stutt(\{x = 4\})$
$(x = 4, \varnothing) \rightarrowtail x = 4$
$(x = 4, \varnothing) \rightarrowtail x = 4$

Figure 3.6: Graph representation of the fixpoint $\mathcal{F}[\![D]\!](p(x))$ in Example 3.1.22.

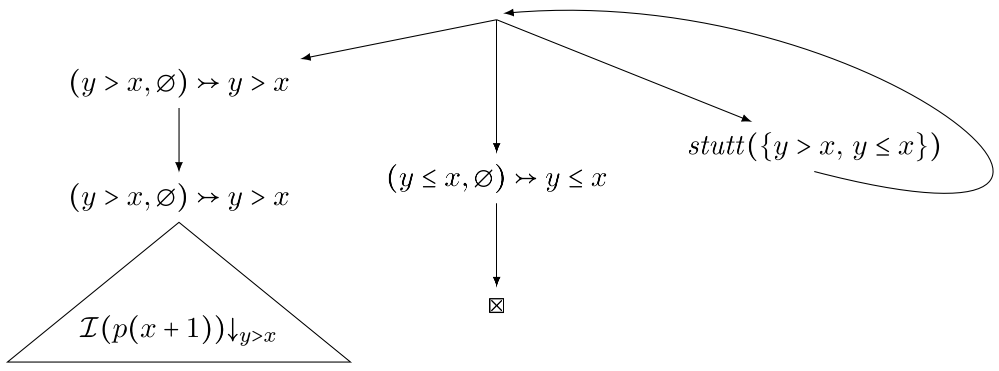

$(y > x, \varnothing) \rightarrowtail y > x$
$stutt(\{y > x,\ y \le x\})$
$(y > x, \varnothing) \rightarrowtail y > x$
$(y \le x, \varnothing) \rightarrowtail y \le x$
$\mathcal{I}(p(x+1))\downarrow_{y>x}$
$\boxtimes$

Figure 3.7: Graph representation for $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ in Example 3.1.23.

$\mathcal{F}[\![D]\!](p(x))$ is graphically represented in Figure 3.6. Note that the application of the propagation operator to the previous iterates removes all the stuttering sequences, and this is the reason because just the first stuttering sequence remains.

Example 3.1.23

Let $D := \{p(x, y) :\!-\ A\}$ where

$$A := \mathsf{ask}(y > x) \rightarrow p(x+1, y) + \mathsf{ask}(y \le x) \rightarrow \mathsf{skip}$$

As usually done in the *tccp* community, we assume that we can use expressions of the form $x+1$ directly in the arguments of a process call. We can simulate this behavior by writing $\exists x' \,(\mathsf{tell}(x' = x+1) \parallel p(x', y))$ instead of $p(x+1, y)$ (but introducing a delay of one time unit). This agent takes two arguments and, if the first is greater or equal than the second, then it stops; otherwise, it performs a recursive call increasing the first argument by one, until it becomes greater or equal to the second one. This process can be combined with other processes to be used as a kind of *timer* since it forces the time passing during a given time interval.

We have

$$\begin{aligned}\mathcal{A}[\![A]\!]_{\mathcal{I}} = {} & \{(y > x, \varnothing) \rightarrowtail y > x \cdot (y > x, \varnothing) \rightarrowtail y > x \cdot r\downarrow_{y>x} \ \mid\ r \in \mathcal{I}(p(x+1, y))\} \sqcup \\ & \{(y \le x, \varnothing) \rightarrowtail y \le x \cdot \boxtimes\} \sqcup\end{aligned}$$

$$\begin{tikzcd}
 & {} \arrow[dl] \arrow[d] \arrow[dr] & \\
(y > x,\emptyset) \rightarrowtail y > x \arrow[d] & (y \le x,\emptyset) \rightarrowtail y \le x \arrow[d] & \mathit{stutt}(\{y > x,\, y \le x\}) \arrow[ul, bend right] \\
(y > x,\emptyset) \rightarrowtail y > x & \boxtimes &
\end{tikzcd}$$

Figure 3.8: Graph representation of $\mathcal{D}[\![D]\!]{\uparrow}1(p(x,y))$ in Example 3.1.23.

$$\begin{tikzcd}
 & {} \arrow[dl] \arrow[d] \arrow[dr] & \\
(y > x,\emptyset) \rightarrowtail y > x \arrow[d] & (y \le x,\emptyset) \rightarrowtail y \le x \arrow[d] & \mathit{stutt}(\{y > x,\, y \le x\}) \arrow[ul, bend right] \\
(y > x,\emptyset) \rightarrowtail y > x \arrow[d] \arrow[dr] & \boxtimes & \\
(y > x+1,\emptyset) \rightarrowtail y > x+1 \arrow[d] & (y = x+1,\emptyset) \rightarrowtail y = x+1 \arrow[d] & \\
(y > x+1,\emptyset) \rightarrowtail y > x+1 & \boxtimes &
\end{tikzcd}$$

Figure 3.9: Graph representation of $\mathcal{D}[\![D]\!]{\uparrow}2(p(x,y))$ in Example 3.1.23.

$$\begin{aligned}
&\{(\mathit{stutt}(\{y > x,\, y \le x\}))^n \cdot (y > x,\emptyset) \rightarrowtail y > x \cdot \\
&\quad (y > x,\emptyset) \rightarrowtail y > x \cdot r{\downarrow}_{y>x} \mid n \in \mathbf{N},\ r \in \mathcal{I}(p(x+1,y))\} \sqcup \\
&\{(\mathit{stutt}(\{y > x,\, y \le x\}))^n \cdot (y \le x,\emptyset) \rightarrowtail y \le x \cdot \boxtimes \ \mid\ n \in \mathbf{N}\} \sqcup \\
&\{\mathit{stutt}(\{y > x,\, y \le x\}) \cdots \mathit{stutt}(\{y > x,\, y \le x\}) \cdots\}
\end{aligned}$$

which is graphically shown in Figure 3.7. For this agent, we have three branches, one for each condition of the choice and one corresponding to the stuttering possibility.

The first iteration of $\mathcal{D}[\![D]\!]$ is

$$\mathcal{D}[\![D]\!]{\uparrow}1 = \begin{cases}
p(x,y) \mapsto & \{(y > x,\emptyset) \rightarrowtail y > x \cdot (y > x,\emptyset) \rightarrowtail y > x\} \sqcup \\
& \{(y \le x,\emptyset) \rightarrowtail y \le x \cdot \boxtimes\} \sqcup \\
& \{(\mathit{stutt}(\{y > x,\, y \le x\}))^n \cdot (y > x,\emptyset) \rightarrowtail y > x \cdot \\
& \quad (y > x,\emptyset) \rightarrowtail y > x \ \mid\ n \in \mathbf{N}\} \sqcup \\
& \{(\mathit{stutt}(\{y > x,\, y \le x\}))^n \cdot (y \le x,\emptyset) \rightarrowtail y \le x \cdot \boxtimes \mid n \in \mathbf{N}\} \sqcup \\
& \{\mathit{stutt}(\{y > x,\, y \le x\}) \cdots \mathit{stutt}(\{y > x,\, y \le x\}) \cdots\}
\end{cases}$$

which is graphically represented in Figure 3.8. Figure 3.9 represents the second iteration $\mathcal{D}[\![D]\!]{\uparrow}2(p(x,y))$, whereas

Figure 3.10 is the graphical representation of $\mathcal{F}[\![D]\!](p(x,y))$. By looking at the semantics, it can be observed that the process stops in one time instant when $y \le x$ and in

$$\begin{tikzcd}
& {} \arrow[dl] \arrow[d] \arrow[dr] & \\
(y > x, \emptyset) \rightarrowtail y > x \arrow[d] & (y \le x, \emptyset) \rightarrowtail y \le x \arrow[d] & stutt(\{y > x,\, y \le x\}) \arrow[ul, bend right=60] \\
(y > x, \emptyset) \rightarrowtail y > x \arrow[d] \arrow[dr] & \boxtimes & \\
(y > x + 1, \emptyset) \rightarrowtail y > x + 1 \arrow[d] & (y = x + 1, \emptyset) \rightarrowtail y = x + 1 \arrow[d] & \\
(y > x + 1, \emptyset) \rightarrowtail y > x + 1 \arrow[d] \arrow[dr] & \boxtimes & \\
(y > x + 2, \emptyset) \rightarrowtail y > x + 2 \arrow[d, dash, dashed, bend right] \arrow[d, dash, dashed, bend left] & (y = x + 2, \emptyset) \rightarrowtail y = x + 2 \arrow[d] & \\
{} & \boxtimes &
\end{tikzcd}$$

Figure 3.10: Graph representation of $\mathcal{F}[\![D]\!](p(x,y))$ in Example 3.1.23.

$1 + 2(y - x)$ time instants otherwise.

Example 3.1.24

Consider the following program process declaration, already introduced in Chapter 2, which models a subsystem of a microwave controller. The underlying constraint system is the (well-known) Herbrand constraint system [41].

$$\begin{array}{l}
\mathit{microwave}(\mathit{Door}, \mathit{Button}, \mathit{Error}) :\!-\ \exists D\, \exists B\, \exists E \\
\quad \big(\ \mathsf{tell}(\mathit{Error} = [\_ \mid E]) \parallel \mathsf{tell}(\mathit{Door} = [\_ \mid D]) \parallel \mathsf{tell}(\mathit{Button} = [\_ \mid B]) \\
\qquad \parallel \mathsf{now}(\mathit{Door} = [\mathit{open} \mid D] \wedge \mathit{Button} = [\mathit{on} \mid B]) \\
\qquad\quad \mathsf{then}\ \big(\ \exists E1\, \mathsf{tell}(E = [1 \mid E1]) \parallel \exists B1\, \mathsf{tell}(B = [\mathit{off} \mid B1])\big) \\
\qquad\quad \mathsf{else}\ \exists E1\, \mathsf{tell}(E = [0 \mid E1]) \\
\qquad \parallel \mathit{microwave}(D, B, E)\big)
\end{array}$$

This process declaration detects if the door is open while the microwave is turned on. In that case, it forces that in the next time instant the microwave is turned-off and it emits an error signal (value 1); otherwise, the agent emits a signal of no error (value 0). Due to the monotonicity of the store, streams are used to model *imperative-style* variables [43]. In the example, the streams *Error*, *Door* and *Button* store the values that the simulated *modifiable variables* get along the computation. The first three tell agents link the future values of the streams with the *future streams* E, D and B. Then, when it is detected a possible *risk* (characterized by the guard of the now agent), the microwave is turned off and an *error* signal is emitted (by the then branch of the conditional agent). The final recursive call restarts the same control at the next time instant.

The fixpoint semantics $\mathcal{F}(\mathit{microwave}(D, B, E))$ is graphically represented in Figure 3.11, where:

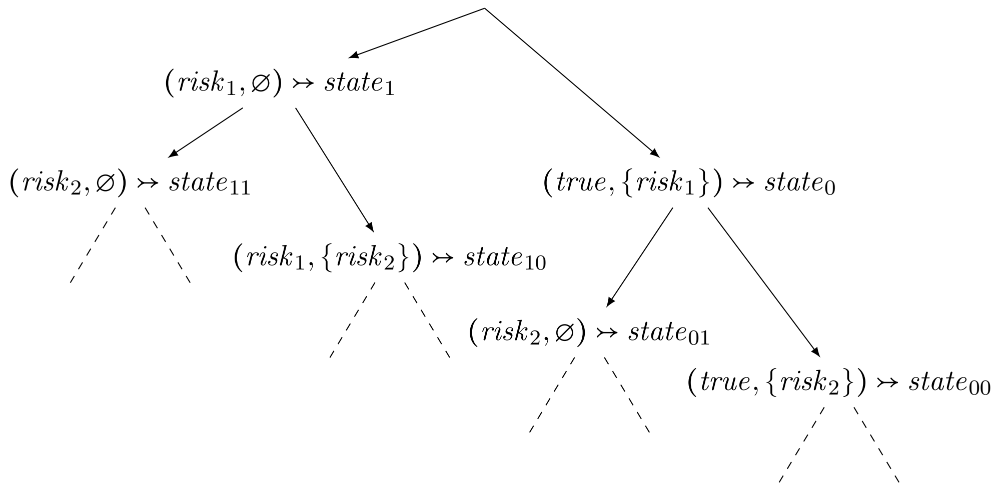

$(risk_1, \varnothing) \rightarrowtail state_1$
$(risk_2, \varnothing) \rightarrowtail state_{11}$
$(risk_1, \{risk_2\}) \rightarrowtail state_{10}$
$(true, \{risk_1\}) \rightarrowtail state_0$
$(risk_2, \varnothing) \rightarrowtail state_{01}$
$(true, \{risk_2\}) \rightarrowtail state_{00}$

Figure 3.11: Tree representation of $\mathcal{F}[\![D]\!](microwave(D,B,E))$ in Example 3.1.24.

$$risk_k := \exists_D\, \exists_B (Door = [\underbrace{open \mid \dots \mid}_{k \text{ times}} D] \wedge Button = [on \mid \underbrace{off \mid on \mid \dots \mid}_{k-1 \text{ times}} B])$$

$$state_{b_1 \dots b_n} := \exists_{E1}\, \exists_D\, \exists_{B1} (Error = [\_ \mid b_1 \mid \dots \mid b_n \mid E1] \wedge Door = [\_ \mid D] \wedge$$
$$Button = [\_ \mid \underbrace{on \mid off \mid \dots}_{\Sigma_{i=1}^{n} b_i \text{ times}} \mid B1])$$

We have coded the indices of stores in the conditional states with a binary number in order to make the figure more readable. It is worth noticing that the stores labeled with $state_b$ where the last digit of b is 1 correspond to states where an error is emitted.

All the conditional sequences in the semantics of this process are infinite sequences. This is consistent with the fact that we are modeling a process that is intended to be active forever (checking whether the risky situation holds). It is worth noticing that this kind of processes can be handled only if the semantics is able to capture infinite computations, which is one of the main features of our proposal.

Full abstraction of $\mathcal{F}$ semantics

In the following we formally prove that our semantics $\mathcal{F}$ is (correct and) fully abstract w.r.t. the small-step operational behavior. To formally link hypothetical computations with real ones, we first need to define an auxiliary operator which, taken an initial store c, instantiates the hypothetical states of a conditional trace r producing the corresponding (real) behavioral timed trace. Intuitively, this operator works by consistently adding to each conditional state the information given by the initial store c, discarding those sequences which falsify conditions.

Definition 3.1.25 (Instantiation operator) *The* instantiation *operator* $\Downarrow : \mathbf{M} \times \mathbf{C} \to \mathbf{C}^*$ *is a partial function defined by structural induction as:* $\epsilon \Downarrow_c := \epsilon$*; otherwise* $r \Downarrow_{false} := false$*; otherwise* $\boxtimes \Downarrow_c := c$*, otherwise*

$$(stutt(\eta^-) \cdot r') \Downarrow_c := c \qquad\qquad if\ \forall c^- \in \eta^- .\, c \nvdash c^-$$

$$(\eta \rightarrowtail d \cdot r')\Downarrow_c := c \cdot (r' \Downarrow_{c \otimes d}) \qquad\qquad \textit{if } c \Vdash \eta \textit{ and } (c \otimes d) \neq \textit{false}$$

We abuse notation by denoting with $R\Downarrow_c$ *the extension of* $\Downarrow_c$ *to* $\mathbf{M}$*:* $R\Downarrow_c := \{r\Downarrow_c \mid r \in R \text{ and } r\Downarrow_c \text{ is defined}\}$*.*

The instantiation operator is consistent w.r.t. the propagation operator (Definition 3.1.7), in the sense that, for any c' that entails c, $r\Downarrow_c = (r\downarrow_{c'})\Downarrow_c$ (as stated formally in Lemma 3.A.6). Moreover, the instantiation operator $\Downarrow$ "distributes" over the parallel composition operator $\bar{\|}$ (Definition 3.1.10) (as stated formally in Lemma 3.A.8).

The key result to prove correctness of $\mathcal{F}$ w.r.t. $\approx_{ss}$ is the following theorem which shows that the small-step behavior of a program P can be determined by instantiation of the semantics $\mathcal{P}[\![P]\!]$.

Theorem 3.1.26 *For each program* P *and each* $c \in \mathbf{C}$*,* $\textit{prefix}(\mathcal{P}[\![P]\!]\Downarrow_c) = \mathcal{B}^{ss}[\![P]\!]_c$*.*

The following theorem is the key result to prove full abstraction of $\mathcal{F}$ w.r.t. $\approx_{ss}$.

Theorem 3.1.27 *Let* P_1, P_2 *be two programs. Then* $\mathcal{P}[\![P_1]\!] = \mathcal{P}[\![P_2]\!]$ *if and only if* $\mathcal{B}^{ss}[\![P_1]\!] = \mathcal{B}^{ss}[\![P_2]\!]$*.*

Proposition 3.1.28 *Let* D_1*,* $D_2 \in \mathbf{D}^{\Pi}_{\mathbf{C}}$*. Then* $D_1 \approx_{\mathcal{F}} D_2$ *if and only if* $\forall A \in \mathbf{A}^{\Pi}_{\mathbf{C}}. \mathcal{P}[\![D_1 \,.\, A]\!] = \mathcal{P}[\![D_2 \,.\, A]\!]$*.*

Correctness and full abstraction is a direct consequence of Theorems 3.1.26 and 3.1.27 and Proposition 3.1.28.

Corollary 3.1.29 (Correctness and full abstraction of $\mathcal{F}$) *Let* D_1*,* $D_2 \in \mathbf{D}^{\Pi}_{\mathbf{C}}$*. Then* $D_1 \approx_{ss} D_2$ *if and only if* $D_1 \approx_{\mathcal{F}} D_2$*.*

3.2 Big-step Semantics

A small-step behavior contains all the details of the computation. However typically only some parts of the execution are considered relevant. So frequently is better to reason only about a specific abstraction of the small-step behavior, instead of dealing with all execution details. In the literature, many authors (like [43]) call *observables* all the abstractions of the small-step behavior of a specific program[4] (including the small-step behavior itself as the degenerate identity abstraction). Moreover, they typically use this same name for the collection of all observables of a set of declarations.

Many other authors use the term *observable property* (or simply observable) for an abstraction function σ which, when applied to the set of traces of a program, delivers the observations of interest. Then the *observation*, or *observable behavior*, of program P is just the application of σ to the traces of P.

We prefer to use the latter nomenclature and, in the sequel, we call *observable behavior of a program Q w.r.t. observable σ* (or simply *σ-observable behavior of Q*) the image $\sigma(\mathcal{B}^{ss}[\![Q]\!])$ and we denote it by $\mathcal{B}^{\sigma}[\![Q]\!]$.

[4] Notice that a *tccp* program is the syntactic correspondent of a (program's) expression of a generic language, while a *tccp* set of declarations is the syntactic correspondent of a program.

The observable property which is usually considered in papers dealing with semantics of *ccp* languages (e.g. see [47]) is the one that collects the input/output pairs of terminating computations, including deadlocked ones. Indeed, using the (original version of the) transition system of Definition 2.2.1, [43] defines the notion of *input-output observable behavior* as $\mathcal{O}^{io}(A) := \{\langle c_0, c_n\rangle \mid \langle A_0, c_0\rangle \to^* \langle A_n, c_n\rangle \not\to\}$. In this definition, there is an implicit reference to a set of declarations D. Since in the sequel we need to state some formal results for two (different) sets of declarations simultaneously, we use the explicit notation $\mathcal{O}^{io}[\![D\,.\,A]\!]$ instead of $\mathcal{O}^{io}(A)$.

As we already mentioned, in *tccp* also infinite computations *must* be considered, for example when we are modeling reactive systems. However, we nevertheless want to be able to distinguish if an input-output pair refers to a finite or infinite computation. Thus, we use *input-output pairs with associated termination mode* of the form $\langle c_0, \mathit{mode}(c_n)\rangle$, where $c_0 \in \mathbf{C}$ is the input store of the computation, $c_n \in \mathbf{C}$ is the output store (which is the *lub* of the stores of the computation) and *mode* is either *fin* or *inf* for finite or infinite computations, respectively. The intuitive idea is that $\langle c_0, \mathit{fin}(c_n)\rangle$ represents all the finite computations that start from the store c_0 and terminate or suspend in a store c_n; and $\langle c_0, \mathit{inf}(l)\rangle$ represents the infinite computations with initial store c_0 and limit constraint l.

Definition 3.2.1 *Given $c, c' \in \mathbf{C}$ such that $c' \vdash c$, an* input-output pair with termination mode *is either $\langle c, \mathit{fin}(c')\rangle$ or $\langle c, \mathit{inf}(c')\rangle$.*

We denote by $\mathbf{IO}$ *the set of input-output pairs with termination mode and by* $\mathbf{IO}$ *the domain $\wp(\mathbf{IO})$, ordered by set inclusion.*

Clearly, $(\mathbf{IO}, \subseteq, \bigcup, \bigcap, \mathbf{IO}, \varnothing)$ is a complete lattice.

Definition 3.2.2 (Input-output behavior of programs) *The* input-output observable *is defined as*

$$io(T) := \big\{\langle c_0, \mathit{fin}(c_n)\rangle \,\big|\, c_0\cdots c_n \in T\big\} \cup \big\{\langle c_0, \mathit{inf}(\otimes_{i\geq 0} c_i)\rangle \,\big|\, c_0\cdots c_n\cdots \in T\big\}.$$

For each $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$ and $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$, the induced input-output behavior *$\mathcal{B}^{io}[\![D\,.\,A]\!]$ is defined as $io(\mathcal{B}^{ss}[\![D\,.\,A]\!])$. We denote by $\approx_{io}$ the equivalence relation between process declarations induced by $\mathcal{B}^{io}$, namely $D_1 \approx_{io} D_2 \iff \forall A \in \mathbf{A}^{\Pi}_{\mathbf{C}}.\ \mathcal{B}^{io}[\![D_1\,.\,A]\!] = \mathcal{B}^{io}[\![D_2\,.\,A]\!]$.*

We denote by π_F the projection which selects just the pairs whose mode is fin and by $\mathbf{IO}_F$ we denote $\pi_F(\mathbf{IO})$. Moreover, we denote by $\mathcal{B}^{io}_F[\![D\,.\,A]\!]$ the finite fragment of $\mathcal{B}^{io}[\![D\,.\,A]\!]$ i.e., $\pi_F(\mathcal{B}^{io}[\![D\,.\,A]\!])$.

Note that, by Definitions 3.1.1 and 3.2.2,

$$\begin{aligned}\mathcal{B}^{io}[\![D\,.\,A_0]\!] = &\big\{\langle c_0, \mathit{fin}(c_n)\rangle \,\big|\, c_0 \in \mathbf{C},\ \langle A_0, c_0\rangle \to^* \langle A_n, c_n\rangle \not\to\big\} \cup \\ &\big\{\langle c_0, \mathit{inf}(\otimes_{i\geq 0} c_i)\rangle \,\big|\, c_0 \in \mathbf{C},\ \langle A_0, c_0\rangle \to \cdots \to \langle A_i, c_i\rangle \to \cdots\big\}\end{aligned}$$

In the sequel, we define an abstract interpretation ([33]) of the small-step semantics $\mathcal{P}[\![D\,.\,A]\!]$ (Definition 3.1.20) which gives $\mathcal{B}^{io}[\![D\,.\,A]\!]$. Then we prove that the finite fragment of this abstraction (i.e., $\mathcal{B}^{io}_F[\![D\,.\,A]\!]$) is essentially isomorphic to $\mathcal{O}^{io}[\![D\,.\,A]\!]$. Actually, there is a negligible difference between $\mathcal{B}^{io}_F[\![D\,.\,A]\!]$ and $\mathcal{O}^{io}[\![D\,.\,A]\!]$ due to the change we made in the definition of the small-step operational semantics. We will state the formal result in Subsection 3.2.2.

To define the semantics modeling the input-output observable as suggested by the abstract interpretation approach (see Section 1.3), we proceed as described in the following.

First, we formalize program properties of interest (in this particular case the input-output behavior) as a Galois Insertion $(\mathbf{M}, \sqsubseteq) \xrightleftharpoons[\alpha]{\gamma} (\mathbf{IO}, \subseteq)$ and then we lift it over interpretations $\mathbf{I} \xrightleftharpoons[\dot\alpha]{\dot\gamma} [\mathbf{PC} \to \mathbf{IO}]$ by function composition as $\dot\alpha(f) = \alpha \circ f$. The best correct (optimal) abstract version of the semantics $\mathcal{D}[\![D]\!]$ is simply obtained as $\mathcal{D}^{\alpha}[\![D]\!] := \dot\alpha \circ \mathcal{D}[\![D]\!] \circ \dot\gamma$. Abstract interpretation theory assures that $\mathcal{F}^{\alpha}[\![D]\!] := \mathit{lfp}(\mathcal{D}^{\alpha}[\![D]\!])$ is the best correct approximation of $\mathcal{F}[\![D]\!]$. Correct because $\alpha(\mathcal{F}[\![D]\!]) \subseteq \mathcal{F}^{\alpha}[\![D]\!]$ and best because it is the minimum (w.r.t. $\subseteq$) of all correct approximations.

3.2.1 Input-output semantics with infinite outcomes

Now we formally define the Galois Insertion which abstracts conditional traces to input-output pairs with termination mode. In the sequel, we denote by $\mathit{last}(s)$ the partial function that, for a non-empty finite sequence s, gives its last element and is otherwise undefined.

Definition 3.2.3 (Input-Output abstraction) *Given any $M \in \mathbf{M}$, we define*

$$\begin{aligned}\alpha_{io}(M) := {} & \{\langle c_0, \mathit{fin}(c_n)\rangle \mid c_0 \in \mathbf{C}, r \in M, \mathit{last}(r{\Downarrow}_{c_0}) = c_n\} \cup \\ & \{\langle c_0, \mathit{inf}(\otimes_{i\geq 0} c_i)\rangle \mid c_0 \in \mathbf{C}, r \in M, r{\Downarrow}_{c_0} = c_0 \dots c_i \dots\}\end{aligned} \tag{3.2.1}$$

$$\begin{aligned}\gamma_{io}(P) := {} & \bigsqcup\{r \in \mathbf{M} \mid \langle c_0, \mathit{fin}(c_n)\rangle \in P, \mathit{last}(r{\Downarrow}_{c_0}) = c_n\} \sqcup \\ & \bigsqcup\{r \in \mathbf{M} \mid \langle c_0, \mathit{inf}(c)\rangle \in P, r{\Downarrow}_{c_0} = c_0 \dots c_i \dots, c = \otimes_{i\geq 0} c_i\}\end{aligned} \tag{3.2.2}$$

We abuse notation and denote with the same symbols the lifting to interpretations, i.e., $\alpha_{io}(\mathcal{I}) := \alpha_{io} \circ \mathcal{I}$, $\gamma_{io}(\mathcal{I}^{\alpha}) := \gamma_{io} \circ \mathcal{I}^{\alpha}$.

$(\mathbf{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}, \{\epsilon\}) \xrightleftharpoons[\alpha_{io}]{\gamma_{io}} (\mathbf{IO}, \subseteq, \bigcup, \bigcap, \mathbf{IO}, \varnothing)$ is a Galois Insertion (as stated formally in Lemma 3.A.9).

The input-output behavior of a program is indeed obtainable by abstraction of its (concrete) semantics.

Proposition 3.2.4 *Let $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$ and $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$. Then, $\alpha_{io}(\mathcal{P}[\![D\,.\,A]\!]) = \mathcal{B}^{io}[\![D\,.\,A]\!]$.*

Now (as anticipated), following the (classical) abstract interpretation approach, we define the optimal abstract version of $\mathcal{D}$ as $\mathcal{D}^{io} := \alpha_{io} \circ \mathcal{D} \circ \gamma_{io}$,[5] and thus the best (possible) correct approximation w.r.t. α_{io} of the semantic function $\mathcal{F}$ is the least fixpoint of $\mathcal{D}^{io}$, i.e., $\mathcal{F}^{io}[\![D]\!] := \mathit{lfp}(\mathcal{D}^{io}[\![D]\!])$. Unfortunately, $\mathcal{F}^{io}[\![D]\!]$ turns out to be very imprecise, mainly because the information contained in the input-output pairs is not enough to keep the synchronization between parallel processes. Indeed, the declarations equivalence induced by $\mathcal{F}^{io}$ is *not correct* w.r.t. $\approx_{io}$ (Definition 3.2.2), since we can have two programs with the same $\mathcal{F}^{io}$ that have different $\mathcal{B}^{io}$, as shown by the following example.

Example 3.2.5

Consider the two sets of declarations $D_1 := \{d_1, d_2\}$ and $D_2 := \{d_1, d_3\}$ where

$$d_1 := p(x, y) :- q(x) \parallel \mathsf{ask}(\mathit{true}) \to \mathsf{now}\ x = 2\ \mathsf{then}\ \mathsf{tell}(y = 0)\ \mathsf{else}\ \mathsf{tell}(y = 1)$$

[5]Although possible, a direct (expanded) definition of $\mathcal{D}^{io}$ is not relevant for our present purposes.

$$d_2 := q(x) :- \mathsf{tell}(x = 2)$$
$$d_3 := q(x) :- \mathsf{ask}(\mathit{true}) \to \mathsf{tell}(x = 2)$$

Clearly, D_2 differs from D_1 just because of the delay in adding the constraint $x = 2$ to the store. This difference shows up in the input-output behavior of $p(x, y)$. Indeed,

$$\begin{aligned}\alpha_{io}(\mathcal{P}[\![D_1 . p(x, y)]\!]) &= \{\langle c, \mathit{fin}(c \wedge x = 2 \wedge y = 0)\rangle \mid c \in \mathbf{L}\}\\ \alpha_{io}(\mathcal{P}[\![D_2 . p(x, y)]\!]) &= \{\langle c, \mathit{fin}(c \wedge y = 0)\rangle \mid c \in \mathbf{L}, c \Rightarrow x = 2\} \cup\\ &\quad\ \{\langle c, \mathit{fin}(c \wedge x = 2 \wedge y = 1)\rangle \mid c \in \mathbf{L}, c \not\Rightarrow x = 2\}\end{aligned}$$

and then (by Proposition 3.2.4) $D_1 \not\approx_{io} D_2$. However, the abstract fixpoint semantics $\mathcal{F}^{io}$ does not distinguish D_1 from D_2. Indeed,

$$\mathcal{F}^{io}[\![D_1]\!] = \mathcal{F}^{io}[\![D_2]\!] = \begin{cases} q(x) \mapsto \{\langle c, \mathit{fin}(c \wedge x = 2)\rangle \mid c \in \mathbf{L}\}\\ p(x,y) \mapsto \{\langle c, \mathit{fin}(c \wedge y = 0)\rangle \mid c \in \mathbf{L}, c \Rightarrow x = 2\}\} \cup\\ \qquad\qquad \{\langle c, \mathit{fin}(c \wedge x = 2 \wedge y = 1)\rangle \mid c \in \mathbf{L}, c \not\Rightarrow x = 2\}\end{cases}$$

Given that $\mathcal{F}^{io}$ is the best possible approximation, this also formally proves that it is not possible to have a correct input-output semantics defined solely on the information provided by the input/output pairs (some more information in denotations is necessarily needed to be correct).

This also formally justifies (a posteriori) why [43] defined $\mathcal{O}^{io}(A)$ as a filter of a more concrete semantics instead of using a direct definition.

3.2.2 Modeling the input-output semantics of [43]

In this section, we formally show that the original input-output semantics of *tccp* $\mathcal{O}^{io}[\![D . A]\!]$ (defined in [43]) is essentially isomorphic to $\mathcal{B}^{io}_F[\![D . A]\!]$ (the finite fragment of the semantics introduced in the previous section).

Theorem 3.2.6 *Let P_1 and P_2 be two tccp programs such that no trace in $\mathcal{P}[\![P_1]\!] \sqcup \mathcal{P}[\![P_2]\!]$ is a failed conditional trace. Then, $\mathcal{O}^{io}[\![P_1]\!] = \mathcal{O}^{io}[\![P_2]\!]$ if and only if $\mathcal{B}^{io}_F[\![P_1]\!] = \mathcal{B}^{io}_F[\![P_2]\!]$.*

This theorem does not hold for *any* pair of *tccp* programs. When none of the programs reaches store *false* (along some execution path), we actually have the same input-output pairs (except for the tag *fin*). However, when the store *false* is reached during a computation, this is no longer necessarily true, as shown by the following example. This explains why we qualify as "essentially isomorphic" the relation between $\mathcal{O}^{io}[\![D . A]\!]$ and $\mathcal{B}^{io}_F[\![D . A]\!]$.

Example 3.2.7

Let $P_1 := D . \mathit{loop}$ and $P_2 := D . \mathsf{tell}(\mathit{false})$, where $D := \{\mathit{loop} :- \mathsf{tell}(\mathit{false}) \parallel \mathit{loop}\}$. We have that $\mathcal{B}^{io}_F[\![P_1]\!] = \mathcal{B}^{io}_F[\![P_2]\!] = \{\langle c, \mathit{fin}(\mathit{false})\rangle\}$ while $\mathcal{O}^{io}[\![P_1]\!] = \varnothing \neq \mathcal{O}^{io}[\![P_2]\!] = \{\langle c, \mathit{false}\rangle\}$.

The difference is due to the change we made in the definition of the small-step operational semantics. More specifically, in the operational semantics that we use (Definition 2.2.1), when the store *false* is reached, we cannot have further transitions. We devised $\to$ in this way to be conform with the original rationale of the *ccp* paradigm. As a consequence, when a sequence computes *false*, it is considered as a failed computation with output *false*. In contrast, in the operational semantics of [43], the transition relation $\to$ does not consider the *false* store as a special case and then it is possible to execute an agent on the *false* store.

Note that, if one is interested, it is straightforward to modify Definition 3.2.3 to compute exactly $\mathcal{O}^{io}[\![P]\!]$.

To conclude, it is interesting to note that $\mathcal{B}^{io}_F[\![P]\!]$ can be equivalently obtained by *first* appropriately filtering the conditional traces and *then* applying the abstraction α_{io}. Formally, given $M \in \mathbf{M}$, let $\pi^{\mathbf{M}}_F(M) \coloneqq \{r \in M \mid r \text{ ends with } \boxtimes \text{ or it contains a stuttering}\}$ and let $\mathbf{M}_F \coloneqq \pi^{\mathbf{M}}_F(\mathbf{M})$. Note that this domain contains only traces such that the application of the $\Downarrow$ operator produces just finite sequences of stores. It is straightforward to prove that the following diagram commutes

$$\begin{tikzcd}
(\mathbf{M}, \sqsubseteq) \arrow[r, "\pi^{\mathbf{M}}_F"] \arrow[d, "\alpha_{io}"] & (\mathbf{M}_F, \sqsubseteq) \arrow[d, "\alpha_{io}"] \\
(\mathbf{IO}, \sqsubseteq) \arrow[r, "\pi_F"] & (\mathbf{IO}_F, \sqsubseteq)
\end{tikzcd}$$

3.3 Related Work

As already stated at the beginning of this chapter, for timed concurrent constraint languages, the presence of

- non-determinism,
- local variables and
- timing constructs which are able to handle negative information

significantly complicates the definition of a fully abstract compositional semantics. In this section, we briefly show the impact of these difficulties when defining appropriate denotational semantics for (timed) concurrent constraint languages.

Most of the defined semantics are inspired in that of *ccp*, and characterize the finite input-output or strongest postcondition observable behaviors. The *strongest postcondition* observable collects the pairs of input-output stores such that the program does not produce additional information, i.e., the input coincides with the output.

In [41], the difficulties for handling nondeterminism and infinite behavior in the *ccp* paradigm were investigated. The authors showed that the presence of nondeterminism and synchronization requires relatively complex structures for the denotational model of (non timed) *ccp* languages. Moreover, infinite behaviors (which become natural in the timed extensions) are an additional complication. Traditionally, solutions to these difficulties have been based on the introduction of restrictions on the language.

In [110] the basic ideas for the definition of appropriate semantics for *ccp* languages are illustrated. More specifically, it is given a model based on observing the resting points of (finite) *ccp* processes. The defined semantics is fully abstract for the determinate fragment of *ccp* (i.e., choice agents have always a single branch). For (finite) nondeterministic processes that are monotonic in nature, a fully abstract semantics is given basing on the observation of ask/tell interactions. In [49], a simple denotational semantics fully abstract w.r.t. the upward-closed observable behavior is defined for *confluent ccp*, which is the subclass of *ccp* programs whose observable behavior does not depend on the chosen (non-deterministic) branch. They also define a correct semantics characterizing the input-output relation of (finite) processes for the *restricted-choice ccp*, which is a confluent sublanguage of *ccp* (syntactically restricted to choice agents where either all the branches have the same guard or the guards are all mutually exclusive). As the basis for a method to prove (partial) correctness of *ccp* programs, in [42] a denotational semantics which characterizes the strongest postcondition is given. The semantics is fully abstract for confluent *ccp*. It is also shown that the strongest postcondition semantics is not compositional w.r.t. the hiding agent.

The introduction of time in the *ccp* paradigm raises even more difficulties, in all different timed languages that have been proposed. As we have shown in Example 3.1.18, due to the partial nature of the constraint system, the combination of the hiding operator with non-determinism can make the language behavior non-monotonic and complicates the definition of a compositional, fully abstract denotational semantics [41, 88]. Based on the deterministic fragment of *ccp*, in [106] the authors defined the *tcc* language and its semantics which is fully abstract just for *hiding free processes*. This restriction allows one to avoid the problem of non-monotonic behaviors.

The *ntcc* language extends *tcc* with non-determinism [87] and, inspired by the elegant model for *ccp* based on closure operators of [110], a denotational semantics for the strongest postcondition is defined. The semantics is fully abstract for *locally-independent* processes, i.e., processes in which the non-monotonic agents do not contain bounded variables (i.e., local variables via the hiding construct). The problem of compositionality in the literature shows up if we try to compute the denotations for $\exists x\, A$ in a compositional way. For example, in [87] what happens is that, if we hide the information about x from the denotations of A, the result is not a strongest postcondition for $\exists x\, A$. More recently, [51] proposed a denotational semantics of the fragment of *ntcc* that excludes the timing construct *unless*.

The *Default tcc* language [107] is an extension of *tcc* that makes use of *default values* in order to model *strong preemption*. It adds to *tcc* language a limited form of negative information handling, with a construct that has to be used under so called *stable assumptions* for the negative information in order to avoid chaotic behaviors (notion borrowed from reactive languages like ESTEREL [12]). This aids to overcome the problem of the non-monotonic behavior since, in some sense, defaults *force* to have the Monotonicity property of Definition 3.1.4. The compositional semantics proposed in [107] is fully abstract for agents which satisfy stable assumptions. Their denotational model associates a condition to the computation which plays a similar role to the first positive condition of our conditional traces (but we can allow more behaviors thanks to the others positive and negative conditions along the trace). However, they show that the hiding operator is not definable compositionally in this specific model since its semantics does not satisfy the local determinacy assumption. In [108] the authors extend this denotational model in order to model also the hiding operator and they show its full abstraction for determinate programs. Although the semantics proposed in [107] and [108] are compositional and fully abstract, they do not cover the difficulty derived from the interaction of non-determinism with hiding operators and time constructs which handle negative information. Furthermore, the *Default tcc* language however has a limited expressive power compared to *tccp* since it is deterministic and does not have process calls (and thus is not Turing complete).

The most recent dialect of timed *ccp* we know, the *utcc* language, was introduced in [90] as an extension of *tcc* for modeling mobility (communication of private names, typically used in security protocols or mobile systems). In [50], a denotational model for *utcc* processes based on a simple domain is defined for data-flow analysis. This semantics is fully abstract only for the monotonic fragment of the language. For the same language, [90] defines a denotational semantics characterizing the input-output behavior of processes. This semantics is fully abstract for the monotonic fragment of *utcc* and is based on temporal formulas.

To conclude, to our knowledge, ours is the only proposal which defines a fully abstract semantics for a *full* non-deterministic dialect of timed *ccp* with "negative" constructs and local variables (having so a non-monotonic behavior).

3.4 Discussion on the results

In this chapter, we have presented a small-step semantics that is fully abstract w.r.t. the *tccp* language behavior and that is suitable to be used as the basis of semantics-based program manipulation techniques such as abstract diagnosis. The task of defining a compositional fully-abstract semantics for the language has shown to be difficult due to the non-monotonic nature of the language, which is a characteristic shared with other concurrent languages of the *ccp* family. However, by defining a more elaborated semantic domain (that uses conditions to model hypothetical computations) and a suitable interpretation of the agents' behavior, we have encompassed these difficulties.

To our knowledge, this is the first fully abstract condensed compositional denotational semantics for a non-deterministic language in the *ccp* family that covers the whole language.

We have also defined a big-step semantics for *tccp* as an abstraction of the small-step one. This semantics collects the *limit* stores of (finite and infinite) computations. We have proven that its fragment for finite computations is precise enough to recover the original input-output semantics of the language [43]. Moreover, we also have proven that it is not possible to have a correct input-output semantics which is defined *solely* on the information provided by the input/output pairs.

As future work, we plan to investigate further on applications of our semantics to obtain novel analysis and verification methods. Moreover, we plan to adapt the ideas presented in this chapter to define appropriate fully-abstract semantics for other concurrent languages of the *ccp* family, such as *ntcc*, *utcc* and *tcc*. These adaptations of the semantics are not immediate, since these languages have significant differences w.r.t. *tccp*, but (given the richness of *tccp* w.r.t. the other languages) we are confident that the required effort will be reasonable. Thanks to this, we will be able to straightforwardly adapt the semantic-based analysis and verification methodology defined for *tccp* to such languages.

3.A Proofs

3.A.1 Proofs of Section 3.1

By construction, we can see that the conditional traces computed by $\mathcal{A}$ always satisfy that the store in a given time instant entails the positive condition. Formally,

Property 3.A.1 *Let $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$, $\mathcal{I} \in \mathbf{I}_{\Pi}$ and $r \in \mathcal{A}[\![A]\!]_{\mathcal{I}}$. For each conditional tuple $(\eta^+, \eta^-) \rightarrowtail a$ occurring in r, $a \vdash \eta^+$.*

Proof.
This property is directly verified by (3.1.7) and (3.1.8) of Definition 3.1.16: when a guard is added to the positive condition, it is also added to the correspondent store, and propagated to the subsequent trace.

There exists a relation between the propagation operator $\downarrow$ and the *lub* $\otimes$ of the constraint system: the consecutive propagation of two constraints $(r\!\downarrow_c)\!\downarrow_{c'}$ is equivalent to $r\!\downarrow_{(c \otimes c')}$.

Lemma 3.A.2 *Let $c, c' \in \mathbf{C}$ and $r \in \mathbf{M}$ such that $(r\!\downarrow_{c'})\!\downarrow_c$ is defined. Then $r\!\downarrow_{(c\otimes c')}$ is defined and $(r\!\downarrow_{c'})\!\downarrow_c = r\!\downarrow_{(c\otimes c')}$.*

Proof.
We proceed by structural induction on r.

$r = \epsilon$ and $r = \boxtimes$ Straightforward.

$r = (\eta^+, \eta^-) \rightarrowtail d \cdot r'$ By hypothesis, $(r\!\downarrow_{c'})\!\downarrow_c$ is defined, thus, $c \gg (\eta^+ \otimes c', \eta^-)$ and $(r'\!\downarrow_{c'})\!\downarrow_c$ is defined. It follows directly that $c \otimes c' \gg (\eta^+, \eta^-)$ and, by inductive hypothesis, $(r'\!\downarrow_{c\otimes c'})$ is defined. Thus, $(r\!\downarrow_{c\otimes c'})$ is defined too.

$$\begin{aligned}
(r\!\downarrow_{c'})\!\downarrow_c =& (((\eta^+, \eta^-) \rightarrowtail d \cdot r')\!\downarrow_{c'})\!\downarrow_c \\
& \quad [\text{by Definition 3.1.7}] \\
=& ((c' \otimes \eta^+, \eta^-) \rightarrowtail c' \otimes d \cdot r'\!\downarrow_{c'})\!\downarrow_c \\
& \quad [\text{by Definition 3.1.7}] \\
=& (c \otimes c' \otimes \eta^+, \eta^-) \rightarrowtail c \otimes c' \otimes d \cdot (r'\!\downarrow_{c'})\!\downarrow_c \\
& \quad [\text{by Inductive Hypothesis}] \\
=& (c \otimes c' \otimes \eta^+, \eta^-) \rightarrowtail c \otimes c' \otimes d \cdot r'\!\downarrow_{c\otimes c'} \\
& \quad [\text{by Definition 3.1.7}] \\
=& r\!\downarrow_{c\otimes c'}
\end{aligned}$$

$r = stutt(\eta^-) \cdot r'$ By hypothesis, $(r\!\downarrow_{c'})\!\downarrow_c$ is defined, thus, for all $c^- \in \eta^-$, $c \nvdash c^-$ and $c' \nvdash c^-$. Furthermore, $(r'\!\downarrow_{c'})\!\downarrow_c$ is defined as well. It follows directly that for all $c^- \in \eta^-$, $c \otimes c' \nvdash c^-$ and, by inductive hypothesis, $(r'\!\downarrow_{c\otimes c'})$ is defined. Thus, $(r\!\downarrow_{c\otimes c'})$ is defined too.

$$\begin{aligned}
(r\!\downarrow_{c'})\!\downarrow_c =& ((stutt(\eta^-) \cdot r')\!\downarrow_{c'})\!\downarrow_c \\
& \quad [\text{by Definition 3.1.7}]
\end{aligned}$$

$$
\begin{aligned}
&=(\mathit{stutt}(\eta^-)\cdot r'{\downarrow_{c'}}){\downarrow_c}\\
&\qquad[\text{by Definition 3.1.7}]\\
&=\mathit{stutt}(\eta^-)\cdot (r'{\downarrow_{c'}}){\downarrow_c}\\
&\qquad[\text{by Inductive Hypothesis}]\\
&=\mathit{stutt}(\eta^-)\cdot r'{\downarrow_{c\otimes c'}}\\
&\qquad[\text{by Definition 3.1.7}]\\
&=r{\downarrow_{c\otimes c'}}
\end{aligned}
$$

There exists a relation between the parallel composition and the operator of propagation as stated by the following lemma.

Lemma 3.A.3 *Let $r_1, r_2 \in \mathbf{M}$ and $c \in \mathbf{C}$ such that $r_1{\downarrow_c} \mathbin{\bar{\parallel}} r_2{\downarrow_c}$ is defined. Then $(r_1 \mathbin{\bar{\parallel}} r_2){\downarrow_c}$ is defined and $r_1{\downarrow_c} \mathbin{\bar{\parallel}} r_2{\downarrow_c} = (r_1 \mathbin{\bar{\parallel}} r_2){\downarrow_c}$.*

Proof.

We proceed by structural induction on r_1. Note that, since $r_1{\downarrow_c} \mathbin{\bar{\parallel}} r_2{\downarrow_c}$ is defined, it follows that $r_1{\downarrow_c}$ and $r_2{\downarrow_c}$ are defined as well.

$r_1 = \epsilon$ (or $r_1 = \boxtimes$) and any r_2 The statement follows directly from Definitions 3.1.7 and 3.1.10.

$r_1 = (\eta_1^+, \eta_1^-) \rightarrowtail d_1 \cdot r_1'$ and $r_2 = (\eta_2^+, \eta_2^-) \rightarrowtail d_2 \cdot r_2'$ Since $r_1{\downarrow_c}$ and $r_2{\downarrow_c}$ are defined, it follows that c is consistent with both $\eta_1 = (\eta_1^+, \eta_1^-)$ and $\eta_2 = (\eta_2^+, \eta_2^-)$, and thus, with $\eta_1 \otimes \eta_2$. We have to distinguish two cases.

$c \otimes d_1 \neq \mathit{false}$ and $c \otimes d_2 \neq \mathit{false}$ By inductive hypothesis, $(r_1' \mathbin{\bar{\parallel}} r_2'){\downarrow_c}$ is defined and, since $c \gg \eta_1 \otimes \eta_2$, $(r_1 \mathbin{\bar{\parallel}} r_2){\downarrow_c}$ is defined as well.

$$
\begin{aligned}
r_1{\downarrow_c} \mathbin{\bar{\parallel}} r_2{\downarrow_c} &=((\eta_1^+, \eta_1^-) \rightarrowtail d_1 \cdot r_1'){\downarrow_c} \mathbin{\bar{\parallel}} ((\eta_2^+, \eta_2^-) \rightarrowtail d_2 \cdot r_2'){\downarrow_c}\\
&\qquad[\text{by Definition 3.1.7}]\\
&=((\eta_1^+ \otimes c, \eta_1^-) \rightarrowtail d_1 \otimes c \cdot r_1'{\downarrow_c}) \mathbin{\bar{\parallel}} ((\eta_2^+ \otimes c, \eta_2^-) \rightarrowtail d_2 \otimes c \cdot r_2'{\downarrow_c})\\
&\qquad[\text{by Definition 3.1.10}]\\
&=(\eta_1^+ \otimes \eta_2^+ \otimes c, \eta_1^- \cup \eta_2^-) \rightarrowtail d_1 \otimes d_2 \otimes c \cdot (r_1'{\downarrow_c} \mathbin{\bar{\parallel}} r_2'{\downarrow_c})\\
&\qquad[\text{by Inductive Hypothesis}]\\
&=(\eta_1^+ \otimes \eta_2^+ \otimes c, \eta_1^- \cup \eta_2^-) \rightarrowtail d_1 \otimes d_2 \otimes c \cdot (r_1' \mathbin{\bar{\parallel}} r_2'){\downarrow_c}\\
&\qquad[\text{by Definition 3.1.7}]\\
&=(r_1 \mathbin{\bar{\parallel}} r_2){\downarrow_c}
\end{aligned}
$$

$c \otimes d_1 = \mathit{false}$ or $c \otimes d_2 = \mathit{false}$ In this case, $r_1{\downarrow_c} \mathbin{\bar{\parallel}} r_2{\downarrow_c}$ reaches the store *false* in one step, as also occurs when we compute $(r_1 \mathbin{\bar{\parallel}} r_2){\downarrow_c}$:

$$
\begin{aligned}
r_1{\downarrow_c} \mathbin{\bar{\parallel}} r_2{\downarrow_c} &=((\eta_1^+, \eta_1^-) \rightarrowtail d_1 \cdot r_1'){\downarrow_c} \mathbin{\bar{\parallel}} ((\eta_2^+, \eta_2^-) \rightarrowtail d_2 \cdot r_2'){\downarrow_c}\\
&\qquad[\text{by Definition 3.1.7 and Definition 3.1.10}]\\
&=(\eta_1^+ \otimes \eta_2^+ \otimes c, \eta_1^- \cup \eta_2^-) \rightarrowtail \mathit{false} \cdot \boxtimes\\
&\qquad[\text{by Definition 3.1.7 and Definition 3.1.10}]\\
&=(r_1 \mathbin{\bar{\parallel}} r_2){\downarrow_c}
\end{aligned}
$$

$r_1 = (\eta_1^+, \eta_1^-) \rightarrowtail d_1 \cdot r_1'$ and $r_2 = stutt(\eta_2^-) \cdot r_2'$ Since $r_1{\downarrow_c}$ and $r_2{\downarrow_c}$ are defined, we have that $c \gg \eta_1$ and $c \nvdash c^-$ for all $c^- \in \eta_2^-$. Therefore, $c \gg (\eta_1^+, \eta_1^- \cup \eta_2^-)$. By inductive hypothesis, $(r_1' \,\bar{\|}\, r_2'){\downarrow_c}$ is defined, thus also $(r_1 \,\bar{\|}\, r_2){\downarrow_c}$ is defined.

$$\begin{aligned}
r_1{\downarrow_c} \,\bar{\|}\, r_2{\downarrow_c} =& ((\eta_1^+, \eta_1^-) \rightarrowtail d_1 \cdot r_1'){\downarrow_c} \,\bar{\|}\, (stutt(\eta_2^-) \cdot r_2'){\downarrow_c}\\
&\quad [\text{by Definition 3.1.7}]\\
=& ((\eta_1^+ \otimes c, \eta_1^-) \rightarrowtail d_1 \otimes c \cdot r_1'{\downarrow_c}) \,\bar{\|}\, (stutt(\eta_2^-) \cdot r_2'{\downarrow_c})\\
&\quad [\text{by Definition 3.1.10}]\\
=& (\eta_1^+ \otimes c, \eta_1^- \cup \eta_2^-) \rightarrowtail d_1 \otimes c \cdot (r_1'{\downarrow_c} \,\bar{\|}\, r_2'{\downarrow_c})\\
&\quad [\text{by Inductive Hypothesis}]\\
=& (\eta_1^+ \otimes c, \eta_1^- \cup \eta_2^-) \rightarrowtail d_1 \otimes c \cdot (r_1' \,\bar{\|}\, r_2'){\downarrow_c}\\
&\quad [\text{by Definition 3.1.7}]\\
=& (r_1 \,\bar{\|}\, r_2){\downarrow_c}
\end{aligned}$$

$r_1 = stutt(\eta_1^-) \cdot r_1'$ and $r_2 = stutt(\eta_2^-) \cdot r_2'$ Since $r_1{\downarrow_c}$ and $r_2{\downarrow_c}$ are defined, we have that c does not entail any constraint in $\eta_1^- \cup \eta_2^-$. By inductive hypothesis, $(r_1' \,\bar{\|}\, r_2'){\downarrow_c}$ is defined, thus, we can conclude that also $(r_1 \,\bar{\|}\, r_2){\downarrow_c}$ is defined.

$$\begin{aligned}
r_1{\downarrow_c} \,\bar{\|}\, r_2{\downarrow_c} =& (stutt(\eta_1^-) \cdot r_1'){\downarrow_c} \,\bar{\|}\, (stutt(\eta_2^-) \cdot r_2'){\downarrow_c}\\
&\quad [\text{by Definition 3.1.7}]\\
=& (stutt(\eta_1^-) \cdot r_1'{\downarrow_c}) \,\bar{\|}\, (stutt(\eta_2^-) \cdot r_2'{\downarrow_c})\\
&\quad [\text{by Definition 3.1.10}]\\
=& \, stutt(\eta_1^- \cup \eta_2^-) \cdot (r_1'{\downarrow_c} \,\bar{\|}\, r_2'{\downarrow_c})\\
&\quad [\text{by Inductive Hypothesis}]\\
=& \, stutt(\eta_1^- \cup \eta_2^-) \cdot (r_1' \,\bar{\|}\, r_2'){\downarrow_c}\\
&\quad [\text{by Definition 3.1.7}]\\
=& (r_1 \,\bar{\|}\, r_2){\downarrow_c}
\end{aligned}$$

An important technical result states that the evaluation function for agents $\mathcal{A}$ is closed under context embedding. A context $C[\,]$ consists in a *tccp* agent with a *hole*, which means that $C[A]$ represents the result of replacing the hole in $C[\,]$ with the agent A.

Lemma 3.A.4 *Let $A_1, A_2 \in \mathbf{A}_{\mathbf{C}}^{\Pi}$ and $\mathcal{I} \in \mathbf{I}$. Then $\mathcal{A}[\![A_1]\!]_{\mathcal{I}} = \mathcal{A}[\![A_2]\!]_{\mathcal{I}}$ if and only if, for all context $C[\,]$, $\mathcal{A}[\![C[A_1]]\!]_{\mathcal{I}} = \mathcal{A}[\![C[A_2]]\!]_{\mathcal{I}}$.*

Proof.

$\Leftarrow$ Directly holds.

$\Rightarrow$ This implication follows from Definition 3.1.16. The evaluation function $\mathcal{A}$ is defined by composition of the semantics of its subagents. In particular, the semantics of both, $C[A_1]$ and $C[A_2]$, is computed from the semantics of A_1 and A_2, respectively. Since A_1 and A_2 are equivalent, then also the semantics of $C[A_1]$ and $C[A_2]$ coincide.

Lemma 3.A.5 *For each $A \in \mathbf{A}_{\mathbf{C}}^{\Pi}$ and each $D \in \mathbf{D}_{\mathbf{C}}^{\Pi}$, $\mathcal{A}[\![A]\!]$ and $\mathcal{D}[\![D]\!]$ are continuos.*

Proof.

Consider $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$ and $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$. To prove the continuity of $\mathcal{A}[\![A]\!]$, we have to verify two properties: monotonicity and finitarity. The continuity of $\mathcal{D}[\![D]\!]$ follows directly from the continuity of $\mathcal{A}[\![A]\!]$ and from Definition 3.1.20.

Monotonicity. It is sufficient to show that for each $\mathcal{I}_1, \mathcal{I}_2 \in \mathbf{I}$ and and for each $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$, $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2 \Rightarrow \mathcal{A}[\![A]\!]_{\mathcal{I}_1} \sqsubseteq \mathcal{A}[\![A]\!]_{\mathcal{I}_2}$. Observe that the only case in which $\mathcal{A}$ depends on the interpretation is the case of the process call.

By definition of $\sqsubseteq$, $\mathcal{I}_1(p(\vec{x})) \sqsubseteq \mathcal{I}_2(p(\vec{x}))$, thus:

$$\begin{aligned}\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{I}_1} &= \bigsqcup\{(true, \varnothing) \rightarrowtail true \cdot r \mid r \in \mathcal{I}_1(p(\vec{x}))\}\\ &\sqsubseteq \bigsqcup\{(true, \varnothing) \rightarrowtail true \cdot r \mid r \in \mathcal{I}_2(p(\vec{x}))\} = \mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{I}_2}\end{aligned}$$

Finitarity. Again, it is sufficient to consider the evaluation function $\mathcal{A}$ for the case of the process call. $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ depends on a finitary subset of $\mathcal{I}$, in particular on the subset regarding $p(\vec{x})$ which is a finitary set of conditional traces closed by prefix.

Lemma 3.A.6 *Let $r \in \mathbf{M}$ and $c, c' \in \mathbf{C}$ such that $c \vdash c'$ and $r{\Downarrow_c}$ is defined. Then $(r{\downarrow_{c'}}){\Downarrow_c}$ is defined and $r{\Downarrow_c} = (r{\downarrow_{c'}}){\Downarrow_c}$.*

Proof.

By hypothesis, $r{\Downarrow_c}$ is defined, thus c is compatible with all the conditions occurring in r. Since $c \vdash c'$, it is easy to notice that also c' is compatible with all the conditions occurring in r, thus $r{\downarrow_{c'}}$ is defined. Then, $(r{\downarrow_{c'}}){\Downarrow_c}$ is defined as well. If $c = false$, by Definition 3.1.25, $r{\Downarrow_{false}} = false = (r{\downarrow_{c'}}){\Downarrow_{false}}$. Otherwise, if $c \neq false$, we proceed by induction on the structure of r.

$r = \epsilon$ and $r = \boxtimes$ The statement follows directly from Definitions 3.1.7 and 3.1.25.

$r = (\eta^+, \eta^-) \rightarrowtail d \cdot r'$ We distinguish three sub-cases.

$d \otimes c \neq false$ Since $c \vdash c'$, it follows that $d \otimes c' \neq false$, thus:

$$\begin{aligned}(r{\downarrow_{c'}}){\Downarrow_c} &= (((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\downarrow_{c'}}){\Downarrow_c}\\ &\qquad [\text{by Definition 3.1.7}]\\ &= ((\eta^+ \otimes c', \eta^-) \rightarrowtail d \otimes c' \cdot r'{\downarrow_{c'}}){\Downarrow_c}\\ &\qquad [\text{by Definition 3.1.25}]\\ &= c \cdot (r'{\downarrow_{c'}}){\Downarrow_{c \otimes d \otimes c'}}\\ &\qquad [\text{by Inductive Hypothesis}]\\ &= c \cdot r'{\Downarrow_{c \otimes d \otimes c'}}\\ &\qquad [\text{since } c \vdash c']\\ &= c \cdot r'{\Downarrow_{c \otimes d}}\end{aligned}$$

By Definition 3.1.25, $r{\Downarrow_c} = ((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\Downarrow_c} = c \cdot r'{\Downarrow_{c \otimes d}}$, thus $r{\Downarrow_c} = (r{\downarrow_{c'}}){\Downarrow_c}$.

$\underline{\boldsymbol{d \otimes c = false} \textbf{ and } \boldsymbol{d \otimes c' \neq false}}\lrcorner$ We have that:

$$\begin{aligned}(r{\downarrow_{c'}}){\Downarrow_c} &= ((\eta^+, \eta^-) \rightarrowtail d \cdot r'{\downarrow_{c'}}){\Downarrow_c} \\ &\qquad [\text{by Definition 3.1.7}] \\ &= ((\eta^+ \otimes c', \eta^-) \rightarrowtail d \otimes c' \cdot r'{\downarrow_{c'}}){\Downarrow_c} \\ &\qquad [\text{by Definition 3.1.25}] \\ &= c \cdot \mathit{false}\end{aligned}$$

By Definition 3.1.25, $r{\Downarrow_c} = ((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\Downarrow_c} = c \cdot \mathit{false}$, thus $r{\Downarrow_c} = (r{\downarrow_{c'}}){\Downarrow_c}$.

$\underline{\boldsymbol{d \otimes c' = false}}\lrcorner$ Since $c \vdash c'$, it follows that $d \otimes c = \mathit{false}$, thus:

$$\begin{aligned}(r{\downarrow_{c'}}){\Downarrow_c} &= (((\eta^+, \eta^-) \rightarrowtail d \cdot r'){\downarrow_{c'}}){\Downarrow_c} \\ &\qquad [\text{by Definition 3.1.7}] \\ &= ((\eta^+ \otimes c', \eta^-) \rightarrowtail \mathit{false} \cdot \boxtimes){\Downarrow_c} \\ &\qquad [\text{by Definition 3.1.25}] \\ &= c \cdot \mathit{false}\end{aligned}$$

By Definition 3.1.25, it follows that $r{\Downarrow_c} = c \cdot \mathit{false} = (r{\downarrow_{c'}}){\Downarrow_c}$.

$\underline{\boldsymbol{r = stutt(\eta^-) \cdot r'}}\lrcorner$ By Definition 3.1.25, it follows that:

$$\begin{aligned}(r{\downarrow_{c'}}){\Downarrow_c} &= ((\mathit{stutt}(\eta^-) \cdot r'){\downarrow_{c'}}){\Downarrow_c} \\ &\qquad [\text{by Definition 3.1.7}] \\ &= (\mathit{stutt}(\eta^-) \cdot r'{\downarrow_{c'}}){\Downarrow_c} \\ &\qquad [\text{by Definition 3.1.25}] \\ &= c\end{aligned}$$

By Definition 3.1.25, $r{\Downarrow_c} = (\mathit{stutt}(\eta^-) \cdot r'){\Downarrow_c} = c$, thus $r{\Downarrow_c} = (r{\downarrow_{c'}}){\Downarrow_c}$.

In order to formulate the following Lemma 3.A.8, we need to introduce the counterpart of $\bar{\|}$ on behavioral timed traces.

Definition 3.A.7 *Let s, s_1, $s_2 \in \mathbf{C}^*$. $\breve{\|} : \mathbf{C}^* \times \mathbf{C}^* \to \mathbf{C}^*$ is defined by structural induction as:*

$$s \mathbin{\breve{\|}} \epsilon := s \qquad \epsilon \mathbin{\breve{\|}} s := s \tag{3.A.1a}$$

$$(c_1 \cdot s_1) \mathbin{\breve{\|}} (c_2 \cdot s_2) := \begin{cases} (c_1 \otimes c_2) \cdot (c_2 \otimes s_1 \mathbin{\breve{\|}} c_1 \otimes s_2) & \mathit{if}\ c_1 \otimes c_2 \neq \mathit{false} \\ \mathit{false} & \mathit{if}\ c_1 \otimes c_2 = \mathit{false} \end{cases} \tag{3.A.1b}$$

where, by abusing notation, $c \otimes (c_1 \cdots c_n)$ denotes $(c \otimes c_1) \cdots (c \otimes c_n)$.

We extend this operator to sets of behavioral timed traces as $S_1 \mathbin{\breve{\|}} S_2 = \{s_1 \mathbin{\breve{\|}} s_2 \mid s_1 \in S_1$ and $s_2 \in S_2\}$.

Lemma 3.A.8 *Let $c \in \mathbf{C}$; $A_1, A_2 \in \mathbf{A}^{\Pi}_{\mathbf{C}}$; $\mathcal{I} \in \mathbf{I}$; $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{I}}$ and $r_2 \in \mathcal{A}[\![A_2]\!]_{\mathcal{I}}$ such that $r_1 \mathbin{\bar{\|}} r_2$, $r_1{\Downarrow_c}$ and $r_2{\Downarrow_c}$ are defined. Then, $(r_1 \mathbin{\bar{\|}} r_2){\Downarrow_c}$ is defined and $r_1{\Downarrow_c} \mathbin{\breve{\|}} r_2{\Downarrow_c} = (r_1 \mathbin{\bar{\|}} r_2){\Downarrow_c}$.*

Proof.

Since both $r_1{\Downarrow_c}$ and $r_2{\Downarrow_c}$ are defined, c satisfies all the conditions in r_1 and r_2. It is easy to notice from Definition 3.1.10 that c satisfies also the conditions of $r_1 \mathbin{\bar{\parallel}} r_2$, thus, $(r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_c}$ is defined as well. We proceed to prove that $r_1{\Downarrow_c} \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = (r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_c}$ by induction on the structure of r_1.

$r_1 = \epsilon$ and any r_2 By Definition 3.1.10, $(r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_c} = (\epsilon \mathbin{\bar{\parallel}} r_2){\Downarrow_c} = r_2{\Downarrow_c}$. By Definition 3.1.25 and by Equation (3.A.1a), we obtain: $r_1{\Downarrow_c} \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = \epsilon \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = r_2{\Downarrow_c}$. Thus, $r_1{\Downarrow_c} \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = (r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_c}$.

$r_1 = \boxtimes$ and any $r_2 \neq \epsilon$ By Definition 3.1.10, $(r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_c} = (\boxtimes \mathbin{\bar{\parallel}} r_2){\Downarrow_c} = r_2{\Downarrow_c}$. By Definition 3.1.25 and by Equation (3.A.1b), $r_1{\Downarrow_c} \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = c \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = r_2{\Downarrow_c}$, since $r_2 \neq \epsilon$. Thus, $r_1{\Downarrow_c} \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = (r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_c}$.

$r_1 = \eta_1 \rightarrowtail d_1 \cdot r'_1$ and $r_2 = \eta_2 \rightarrowtail d_2 \cdot r'_2$

$d_1 \otimes d_2 \neq \mathit{false}$

$$\begin{aligned}
(r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_c} &= ((\eta_1 \rightarrowtail d_1 \cdot r'_1) \mathbin{\bar{\parallel}} (\eta_2 \rightarrowtail d_2 \cdot r'_2)){\Downarrow_c} \\
&\qquad [\text{by Definition 3.1.10}] \\
&= (\eta_1 \otimes \eta_2 \rightarrowtail d_1 \otimes d_2 \cdot (r'_1{\downarrow_{d_2}} \mathbin{\bar{\parallel}} r'_2{\downarrow_{d_1}})){\Downarrow_c} \\
&\qquad [\text{by Definition 3.1.25}] \\
&= c \cdot (r'_1{\downarrow_{d_2}} \mathbin{\bar{\parallel}} r'_2{\downarrow_{d_1}}){\Downarrow_{c \otimes d_1 \otimes d_2}} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&= c \cdot ((r'_1{\downarrow_{d_2}}){\Downarrow_{c \otimes d_1 \otimes d_2}} \mathbin{\breve{\parallel}} (r'_2{\downarrow_{d_1}}){\Downarrow_{c \otimes d_1 \otimes d_2}}) \\
&\qquad [\text{by Lemma 3.A.6}] \\
&= c \cdot (r'_1{\Downarrow_{c \otimes d_1 \otimes d_2}} \mathbin{\breve{\parallel}} r'_2{\Downarrow_{c \otimes d_1 \otimes d_2}}) \\
&\qquad [d_1 \text{ (resp. } d_2\text{) is entailed by the stores in } r'_1 \text{ (resp. } r'_2\text{)}] \\
&= c \cdot (r'_1{\Downarrow_{c \otimes d_1}} \mathbin{\breve{\parallel}} r'_2{\Downarrow_{c \otimes d_2}}) \\
&\qquad [\text{by Equation (3.A.1b)}] \\
&= (c \cdot r'_1{\Downarrow_{c \otimes d_1}}) \mathbin{\breve{\parallel}} (c \cdot r'_2{\Downarrow_{c \otimes d_2}})
\end{aligned}$$

By Definition 3.1.25, $r_1{\Downarrow_c} \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = (c \cdot r'_1{\Downarrow_{c \otimes d_1}}) \mathbin{\breve{\parallel}} (c \cdot r'_2{\Downarrow_{c \otimes d_2}})$; therefore, we conclude $r_1{\Downarrow_c} \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = (r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_c}$.

$d_1 \otimes d_2 = \mathit{false}$

$$\begin{aligned}
(r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_c} &= ((\eta_1 \rightarrowtail d_1 \cdot r'_1) \mathbin{\bar{\parallel}} (\eta_2 \rightarrowtail d_2 \cdot r'_2)){\Downarrow_c} \\
&\qquad [\text{by Definition 3.1.10}] \\
&= (\eta_1 \otimes \eta_2 \rightarrowtail \mathit{false} \cdot \boxtimes){\Downarrow_c} \\
&\qquad [\text{by Definition 3.1.25}] \\
&= c \cdot \mathit{false}
\end{aligned}$$

By Definition 3.1.25 and by Equation (3.A.1b), $r_1{\Downarrow_c} \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = c \cdot \mathit{false}$, thus $r_1{\Downarrow_c} \mathbin{\breve{\parallel}} r_2{\Downarrow_c} = (r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_c}$.

$\boldsymbol{r_1 = \eta_1 \rightarrowtail d_1 \cdot r_1'}$ **and** $\boldsymbol{r_2 = stutt(\eta_2^-) \cdot r_2'}$

$$\begin{aligned}
(r_1 \mathbin{\bar{\|}} r_2)\Downarrow_c &= ((\eta_1 \rightarrowtail d_1 \cdot r_1') \mathbin{\bar{\|}} (stutt(\eta_2^-) \cdot r_2'))\Downarrow_c \\
&\qquad [\text{by Definition 3.1.10}] \\
&= ((\eta_1^+, \eta_1^- \cup \eta_2^-) \rightarrowtail d_1 \cdot (r_1' \mathbin{\bar{\|}} r_2'{\downarrow_{d_1}}))\Downarrow_c \\
&\qquad [\text{by Definition 3.1.25}] \\
&= c \cdot (r_1' \mathbin{\bar{\|}} r_2'{\downarrow_{d_1}})\Downarrow_{c \otimes d_1} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&= c \cdot (r_1'\Downarrow_{c \otimes d_1} \mathbin{\breve{\|}} (r_2'{\downarrow_{d_1}})\Downarrow_{c \otimes d_1}) \\
&\qquad [\text{by Lemma 3.A.6}] \\
&= c \cdot (r_1'\Downarrow_{c \otimes d_1} \mathbin{\breve{\|}} r_2'\Downarrow_{c \otimes d_1}) \\
&\qquad [\text{by Equation (3.A.1b)}] \\
&= (c \cdot r_1'\Downarrow_{c \otimes d_1}) \mathbin{\breve{\|}} (c \cdot r_2'\Downarrow_c)
\end{aligned}$$

By Definition 3.1.25, $r_1\Downarrow_c \mathbin{\breve{\|}} r_2\Downarrow_c = (c \cdot r_1'\Downarrow_{c \otimes d_1}) \mathbin{\breve{\|}} (c \cdot r_2'\Downarrow_c)$, thus $r_1\Downarrow_c \mathbin{\breve{\|}} r_2\Downarrow_c = (r_1 \mathbin{\bar{\|}} r_2)\Downarrow_c$.

$\boldsymbol{r_1 = stutt(\eta_1^-) \cdot r_1'}$ **and** $\boldsymbol{r_2 = stutt(\eta_2^-) \cdot r_2'}$

$$\begin{aligned}
(r_1 \mathbin{\bar{\|}} r_2)\Downarrow_c &= ((stutt(\eta_1^-) \cdot r_1') \mathbin{\bar{\|}} (stutt(\eta_2^-) \cdot r_2'))\Downarrow_c \\
&\qquad [\text{by Definition 3.1.10}] \\
&= (stutt(\eta_1^- \cup \eta_2^-) \cdot (r_1' \mathbin{\bar{\|}} r_2'))\Downarrow_c \\
&\qquad [\text{by Definition 3.1.25}] \\
&= c \cdot (r_1' \mathbin{\bar{\|}} r_2')\Downarrow_c \\
&\qquad [\text{by Inductive Hypothesis}] \\
&= c \cdot (r_1'\Downarrow_c \mathbin{\breve{\|}} r_2'\Downarrow_c) \\
&\qquad [\text{by Equation (3.A.1b)}] \\
&= (c \cdot r_1'\Downarrow_c) \mathbin{\breve{\|}} (c \cdot r_2'\Downarrow_c)
\end{aligned}$$

By Definition 3.1.25, $r_1\Downarrow_c \mathbin{\breve{\|}} r_2\Downarrow_c = (c \cdot r_1'\Downarrow_c) \mathbin{\breve{\|}} (c \cdot r_2'\Downarrow_c)$, thus $r_1\Downarrow_c \mathbin{\breve{\|}} r_2\Downarrow_c = (r_1 \mathbin{\bar{\|}} r_2)\Downarrow_c$.

Theorem 3.1.26. *For each program P and each $c \in \mathbf{C}$, prefix$(\mathcal{P}[\![P]\!]\Downarrow_c) = \mathcal{B}^{ss}[\![P]\!]_c$.*

Proof.

Let $d \in \mathbf{C}$ and $P = D.A$ with $D \in \mathbf{D}_{\mathbf{C}}^{\Pi}$ and $A \in \mathbf{A}_{\mathbf{C}}^{\Pi}$, we proceed by structural induction on A.

$\boldsymbol{A = \mathsf{skip}}$ The proof in this case is straightforward.

$$prefix(\mathcal{A}[\![\mathsf{skip}]\!]_{\mathcal{F}[\![D]\!]})\Downarrow_d = prefix(\{\boxtimes\})\Downarrow_d = \{\epsilon,\, d\} = \mathcal{B}^{ss}[\![D \,.\, \mathsf{skip}]\!]_d$$

$\boldsymbol{A = \mathsf{tell}(c)}$

$$\begin{aligned}
prefix((\mathcal{A}[\![\mathsf{tell}(c)]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) &= prefix((true, \varnothing) \rightarrowtail c \cdot \boxtimes)\Downarrow_d) \\
&= prefix(d \cdot (d \otimes c)) \\
&= \mathcal{B}^{ss}[\![D \,.\, \mathsf{tell}(c)]\!]_d
\end{aligned}$$

$A = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i$ We prove the two directions separately.

$\subseteq$ We show that, given a conditional trace $r \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$, it holds that $\forall d \in \mathbf{C}.\, \mathit{prefix}(r{\Downarrow_d}) \subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. We have to distinguish two cases.

$r = (c_j, \varnothing) \rightarrowtail c_j \cdot r_j{\downarrow_{c_j}}$ with $1 \le j \le n$ By (3.1.7) it follows that $r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}$. In case $r{\Downarrow_d}$ is not defined (i.e., $d \nvdash c_j$), $\mathit{prefix}(r{\Downarrow_d}) = \varnothing \subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. Otherwise, if $r{\Downarrow_d}$ is defined, we have that $d \vdash c_j$ and $(r_j{\downarrow_{c_j}}){\Downarrow_{d\otimes c_j}}$ is defined too. We distinguish two sub-cases.

$d \ne \mathit{false}$ In this case we have:

$$\begin{aligned}
&\mathit{prefix}(r{\Downarrow_d})\\
&= \mathit{prefix}(\{((c_j,\varnothing) \rightarrowtail c_j \cdot r_j{\downarrow_{c_j}}){\Downarrow_d} \mid 1 \le j \le n,\ r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\text{by Definition 3.1.25}]\\
&= \mathit{prefix}(\{d \cdot (r_j{\downarrow_{c_j}}){\Downarrow_{d\otimes c_j}} \mid 1 \le j \le n,\ r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\text{by Lemma 3.A.6 and since } d \vdash c_j]\\
&= \mathit{prefix}(\{d \cdot r_j{\Downarrow_d} \mid 1 \le j \le n,\ r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\text{by Equation (3.1.1)}]\\
&= \{\epsilon,\ d\} \cup \{d \cdot s \mid 1 \le j \le n,\ s \in \mathit{prefix}(\mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d})\}\\
&\qquad [\text{by Inductive Hypothesis}]\\
&\subseteq \{\epsilon,\ d\} \cup \{d \cdot s \mid 1 \le j \le n,\ s \in \mathcal{B}^{ss}[\![D\,.\,A_j]\!]_d\}
\end{aligned}$$

The element ϵ directly belongs to $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. Since $d \vdash c_j$, also d belongs to $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$ (at least one step is performed in the computation). Finally, the set $\{d \cdot s \mid 1 \le j \le n,\ \in \mathcal{B}^{ss}[\![D\,.\,A_j]\!]_d\}$ is also contained in $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. In particular, following Rule **R2**, the agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i$ (executed with a store d that entails one of the guards, e.g. c_j) behaves, in the next time instant, as the corresponding agent A_j over the store (which is not modified in that step).

$d = \mathit{false}$ By definition of $\Downarrow$ (3.1.25), we have that $\mathit{prefix}(r{\Downarrow_{\mathit{false}}}) = \{\epsilon, \mathit{false}\}$ which corresponds to the set $\mathcal{B}^{ss}[\![D.A]\!]_{\mathit{false}}$ since the transition relation $\to$ is not defined for the configuration $\langle A, \mathit{false}\rangle$.

$r = \mathit{stutt}(\{c_1, \dots, c_n\}) \cdot r'$ By (3.1.7), we have that $r' \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$ and for all $1 \le j \le n$, $c_j \ne \mathit{true}$. In case $r{\Downarrow_d}$ is not defined (i.e., it exists $1 \le j \le n$ such that $d \vdash c_j$), $\mathit{prefix}(r{\Downarrow_d}) = \varnothing \subseteq \mathcal{B}^{ss}[\![D.A]\!]_d$. Otherwise, if $r{\Downarrow_d}$ is defined then $\mathit{prefix}(r{\Downarrow_d}) = \{\epsilon,\ d\} \subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.

$\supseteq$ For each $d \in \mathbf{C}$, it exists a conditional trace $r \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$ such that $\mathit{prefix}(r{\Downarrow_d}) \supseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. There are three cases to be considered.

d does not satisfy any guard This means that for all $1 \le j \le n$, $d \nvdash c_j$; then, the small-step behavior is $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d = \{\epsilon,\ d\}$. Thus, it exists a conditional trace $r \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$ such that $r = \mathit{stutt}(\{c_1, \dots, c_n\}) \cdot r'$ with $r' \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Moreover, by Definition 3.1.25 and Definition 3.1.1, it follows that $\mathit{prefix}(r{\Downarrow_d}) = \{\epsilon,\ d\} \supseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.

there exists c_j such that $d \vdash c_j$ and $d \neq false$ In this case, one of the conditional traces computed by the semantics evaluation function $\mathcal{A}$ is $r = (c_j, \varnothing) \rightarrowtail c_j \cdot r_j{\downarrow_{c_j}}$ with $r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}$. Then, we have:

$$\begin{aligned}
prefix(r{\Downarrow_d}) &= prefix(((c_j,\varnothing) \rightarrowtail c_j \cdot r_j{\downarrow_{c_j}}){\Downarrow_d}) \\
&\qquad [\text{by Definition 3.1.25}] \\
&= prefix(\{d \cdot (r_j{\downarrow_{c_j}}){\Downarrow_{d \otimes c_j}} \mid r_j \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Lemma 3.A.6 and since } d \vdash c_j] \\
&= prefix(\{d \cdot r_j{\Downarrow_d} \mid r_j{\Downarrow_d} \in \mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}\}) \\
&\qquad [\text{by Equation (3.1.1)}] \\
&= \{\epsilon, d\} \cup \{d \cdot s \mid s \in prefix(\mathcal{A}[\![A_j]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d})\} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\supseteq \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathcal{B}^{ss}[\![D \,.\, A_j]\!]_d\} \\
&\qquad [\text{by Rule } \mathbf{R2}] \\
&\supseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d
\end{aligned}$$

$d = false$ In this case we have:

$$\begin{aligned}
prefix(r{\Downarrow_{false}}) &= prefix(((c_j,\varnothing) \rightarrowtail c_j \cdot r_j{\downarrow_{c_j}}){\Downarrow_{false}}) \\
&\qquad [\text{by Definition 3.1.25}] \\
&= \{\epsilon, false\} \\
&\qquad [\text{by Definition 3.1.1}] \\
&\supseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_{false}
\end{aligned}$$

Therefore, we can conclude that $prefix(\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}) = \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

A = now c then A_1 else A_2 We prove the two directions independently. We abbreviate the conditional agent and call it A ($A :=$ now c then A_1 else A_2).

$\subseteq$ We show that $\forall d \in \mathbf{C}$. $prefix(\mathcal{A}[\![\text{now } c \text{ then } A_1 \text{ else } A_2]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}) \subseteq \mathcal{B}^{ss}[\![D \,.\, \text{now } c \text{ then } A_1 \text{ else } A_2]\!]_d$. There are seven possible cases, one for each type of trace r in (3.1.8).

$r = (c,\varnothing) \rightarrowtail c \cdot \boxtimes$ By (3.1.8) we have that $\boxtimes \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$, which means, by Definition 3.1.16, that A_1 = skip. We consider now the three possible cases:

$d \vdash c$ and $d \neq false$ It is straightforward that $prefix(r{\Downarrow_d}) = prefix(d \cdot d) = \{\epsilon,\ d,\ d \cdot d\}$. On the behavioral part, we know from Rule **R4** that the observable of A is the set of all prefixes of $d \cdot d$, so we can conclude $prefix(r{\Downarrow_d}) \subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

$d = false$ The small-step behavior is $\mathcal{B}^{ss}[\![D \,.\, A]\!]_{false} = \{\epsilon, false\}$. Since $false \vdash c$ it is straightforward that $prefix(r{\Downarrow_{false}}) = \{\epsilon, false\} = \mathcal{B}^{ss}[\![D \,.\, A]\!]_{false}$.

$d \nvdash c$ Then the application of $\Downarrow_d$ to the agent semantics does not compute any behavioral timed trace. Therefore, $prefix(r{\Downarrow_d}) = \varnothing \subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

$\underline{\boldsymbol{r = (\eta^+ \otimes c, \eta^-) \rightarrowtail a \otimes c \cdot r'{\downarrow_c}}}\lrcorner$ From (3.1.8) it follows that $(\eta^+, \eta^-) \rightarrowtail a \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$, d being compatible with all the conditions occurring in r', $a \otimes c \neq \mathit{false}$ and $\forall h^- \in \eta^- . \eta^+ \otimes c \nvdash h^-$.

In case $r{\Downarrow_d}$ is not defined (i.e., $d \nVdash (\eta^+ \otimes c, \eta^-)$ or when d is not compatible with some condition occurring in r'), we have that $\mathit{prefix}(r{\Downarrow_d}) = \varnothing$ which is directly included in $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

Otherwise, if $r{\Downarrow_d}$ is defined, it follows that $d \Vdash (\eta^+ \otimes c, \eta^-)$. This implies that $d \vdash c$ since c belongs to the positive condition. Under these conditions, we have:

$$\begin{aligned}
&\mathit{prefix}(r{\Downarrow_d}) = \\
&= \mathit{prefix}(\{((\eta^+ \otimes c, \eta^-) \rightarrowtail a \otimes c \cdot r'{\downarrow_c}){\Downarrow_d} \mid (\eta^+, \eta^-) \rightarrowtail a \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.1.25}] \\
&= \mathit{prefix}(\{d \cdot (r'{\downarrow_c}){\Downarrow_{d\otimes a\otimes c}} \mid d \cdot r'{\Downarrow_{d\otimes a}} \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}\}) \\
&\qquad [\text{by Lemma 3.A.6 since } d \vdash c] \\
&= \mathit{prefix}(\{d \cdot r'{\Downarrow_{d\otimes a}} \mid d \cdot r'{\Downarrow_{d\otimes a}} \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}\}) \\
&= \mathit{prefix}(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}) \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\subseteq \mathcal{B}^{ss}[\![D \,.\, A_1]\!]_d \\
&\qquad [\text{by Rule } \mathbf{R3}] \\
&\subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d
\end{aligned}$$

$\underline{\boldsymbol{r = (\eta^+ \otimes c, \eta^-) \rightarrowtail \mathit{false} \cdot \boxtimes}}\lrcorner$ We consider two possible cases:

$\underline{\boldsymbol{d \Vdash (\eta^+ \otimes c, \eta^-)}}\lrcorner$ This implies that $d \vdash c$. Under these conditions, we get:

$$\begin{aligned}
&\mathit{prefix}(r{\Downarrow_d}) \\
&= \mathit{prefix}(((\eta^+ \otimes c, \eta^-) \rightarrowtail \mathit{false} \cdot \boxtimes){\Downarrow_d}) \\
&\qquad [\text{by Definition 3.1.25}] \\
&= \{\epsilon, \mathit{false}\} \\
&\qquad [\text{by Definition 3.1.1}] \\
&\subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d
\end{aligned}$$

$\underline{\boldsymbol{d \nVdash (\eta^+ \otimes c, \eta^-)}}\lrcorner$ In this case $\mathit{prefix}(r{\Downarrow_d}) = \varnothing$ which is directly included in $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

$\underline{\boldsymbol{r = (c, \eta^-) \rightarrowtail c \cdot r'}}\lrcorner$ In case $r{\Downarrow_d}$ is not defined (i.e., $d \nVdash (c, \eta^-)$ or also when d is not compatible with some condition occurring in r') we have that $\mathit{prefix}(r{\Downarrow_d}) = \varnothing \subseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$. Otherwise, by (3.1.8), $\mathit{stutt}(\eta^-) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and d is compatible with all the conditions occurring in r'. We have to consider two sub-cases.

$\underline{\boldsymbol{d \neq \mathit{false}}}\lrcorner$ In this case we have:

$$\begin{aligned}
&\mathit{prefix}(r{\Downarrow_d}) = \\
&= \mathit{prefix}(\{((c, \eta^-) \rightarrowtail c \cdot r'){\Downarrow_d} \mid \mathit{stutt}(\eta^-) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.1.25}]
\end{aligned}$$

$$
\begin{aligned}
&= \mathit{prefix}(\{d \cdot r'\Downarrow_{d\otimes c} \mid \mathit{stutt}(\eta^-) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\,\text{since } d \vdash c\,]\\
&= \mathit{prefix}(\{d \cdot r'\Downarrow_{d} \mid \mathit{stutt}(\eta^-) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\,\text{by Definition 3.1.16}\,]\\
&= \mathit{prefix}(\{d \cdot r'\Downarrow_{d} \mid r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\,\text{by Equation (3.1.1)}\,]\\
&= \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathit{prefix}(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)\}\\
&\qquad [\,\text{by Inductive Hypothesis}\,]\\
&\subseteq \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathcal{B}^{ss}[\![D\,.\,A_1]\!]_d\}\\
&\qquad [\,\text{by Rule } \mathbf{R4}\,]\\
&\subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d
\end{aligned}
$$

The fourth step follows from the definition of the semantics $\mathcal{A}$ (Definition 3.1.16). The construct *stutt* is introduced only by an ask agent. Thus, we know that A_1 is an ask agent. The Equation (3.1.8), states that $\mathit{stutt}(\eta^-)$ is always followed by a conditional trace which belongs to the semantics of the ask, which can be reduced to say that r' belongs to $\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$.

$\underline{\boldsymbol{d = \mathit{false}}}_\lrcorner$ In this case we have that $\mathit{prefix}(r\Downarrow_{\mathit{false}}) = \{\epsilon, \mathit{false}\}$ which corresponds to the behavior $\mathcal{B}^{ss}[\![D\,.\,A]\!]_{\mathit{false}}$ since the transition relation $\rightarrow$ is not defined for the agent A starting with store *false*.

$\underline{\boldsymbol{r = (\mathit{true}, \{c\}) \rightarrowtail \mathit{true} \cdot \boxtimes}}_\lrcorner$ By (3.1.8), $\boxtimes \in \mathcal{A}[\![A_2]\!]_{\mathcal{I}}$. By Definition 3.1.16, it follows that A_2 is a skip agent. We consider two sub-cases.

$\underline{\boldsymbol{d \nvdash c}}_\lrcorner$ It is straightforward that $\mathit{prefix}(r\Downarrow_d) = \mathit{prefix}(d \cdot d) = \{\epsilon,\ d,\ d \cdot d\}$. From Rule **R6**, we know that the observable of the agent A consists of the set of all prefixes of $d \cdot d$. Therefore, $\mathit{prefix}(r\Downarrow_d) \subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.

$\underline{\boldsymbol{d \vdash c}}_\lrcorner$ In this case $r\Downarrow_d$ does not compute any trace because d does not satisfy the condition, thus $\mathit{prefix}(r\Downarrow_d) = \varnothing \subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.

$\underline{\boldsymbol{r = (\eta^+, \eta^- \cup \{c\}) \rightarrowtail c' \cdot r'}}_\lrcorner$ In case $r\Downarrow_d$ is not defined (i.e., $d \nVdash (\eta^+, \eta^- \cup \{c\})$), $\mathit{prefix}(r\Downarrow_d) = \varnothing$, which is directly contained in $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.
Otherwise, if $r\Downarrow_d$ it follows that $(\eta^+, \eta^-) \rightarrowtail c' \cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ and $c' \nvdash c$. If $d \Vdash (\eta^+, \eta^- \cup \{c\})$, we know also that $d \nvdash c$. Under these conditions, we have:

$$
\begin{aligned}
&\mathit{prefix}(\{r\Downarrow_d)\\
&= \mathit{prefix}(\{((\eta^+, \eta^- \cup \{c\}) \rightarrowtail c' \cdot r')\Downarrow_d \mid (\eta^+, \eta^-) \rightarrowtail c' \cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\,\text{by Definition 3.1.25}\,]\\
&= \mathit{prefix}(\{d \cdot r'\Downarrow_{d\otimes c'} \mid d \cdot r'\Downarrow_{d\otimes c'} \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d\})\\
&= \mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)\\
&\qquad [\,\text{by Inductive Hypothesis}\,]\\
&\subseteq \mathcal{B}^{ss}[\![D\,.\,A_2]\!]_d\\
&\qquad [\,\text{by Rule } \mathbf{R5}\,]
\end{aligned}
$$

$\subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$

$\underline{\boldsymbol{r = (true, \eta^- \cup \{c\}) \rightarrowtail true \cdot r'}}\lrcorner$ By (3.1.8), we have that $stutt(\eta^-)\cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$. In case $r\Downarrow_d$ is not defined (i.e., $d \not\Vdash (true, \eta^- \cup \{c\})$), $prefix(r\Downarrow_d) = \varnothing$, which is directly contained in $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.
Otherwise, if $r\Downarrow_d$ is defined it follows that $d \Vdash (true, \eta^- \cup \{c\})$. Then, we have:

$$
\begin{aligned}
& prefix(r\Downarrow_d) \\
&= prefix(\{((true, \eta^- \cup \{c\}) \rightarrowtail true \cdot r')\Downarrow_d \mid stutt(\eta^-)\cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.1.25}] \\
&= prefix(\{d \cdot r'\Downarrow_d \mid stutt(\eta^-)\cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Definition 3.1.16}] \\
&= prefix(\{d \cdot r'\Downarrow_d \mid r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\}) \\
&\qquad [\text{by Equation (3.1.1)}] \\
&= \{\epsilon, d\} \cup \{d \cdot s \mid s \in prefix(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)\} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\subseteq \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathcal{B}^{ss}[\![D\,.\,A_2]\!]_d\} \\
&\qquad [\text{by Rule } \mathbf{R6}] \\
&\subseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d
\end{aligned}
$$

The third step can be done since each construct $stutt(\eta^-)$ is introduced by a choice agent, and Equation (3.1.7) states that it is always followed by a conditional trace r' belonging recursively to the semantics of A_2.

$\supseteq$ We have four cases, one for each rule defining the operational semantics for the conditional agent in Figure 2.2.

$\underline{\textbf{Rule R3}}\lrcorner$ Let us recall the conditions to apply Rule **R3**: it must occur $\langle A_1,\, d\rangle \rightarrow \langle A_1',\, d'\rangle$ and $d \vdash c$. In this case, we have that $\mathcal{B}^{ss}[\![D.A]\!]_d = \mathcal{B}^{ss}[\![D.A_1]\!]_d$. By inductive hypothesis, we know that $prefix(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) \supseteq \mathcal{B}^{ss}[\![D\,.\,A_1]\!]_d$, thus also $prefix(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) \supseteq \mathcal{B}^{ss}[\![D\,.\,A]\!]_d$. Next, we prove the inclusion $prefix(\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d) \supseteq prefix(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]})\Downarrow_d$. We proceed by induction on the structure of a generic $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ in order to find $r \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$ such that $prefix(r_1\Downarrow_d) \subseteq prefix(r\Downarrow_d)$.

$\underline{r_1 = \boxtimes}\lrcorner$ By (3.1.8), $r = (c, \varnothing) \rightarrowtail c \cdot \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. We know that $\boxtimes\Downarrow_d = d$ and $r\Downarrow_d = d \cdot (d \otimes c) = d \cdot d$, since $d \vdash c$. It is easy to see that the prefixes of d are all included in the prefixes of $d \cdot d$.

$\underline{\boldsymbol{r_1 = (\eta^+, \eta^-) \rightarrowtail c' \cdot r'}}\lrcorner$ By definition, $r = (\eta^+ \otimes c, \eta^-) \rightarrowtail c' \otimes c \cdot r'\downarrow_c \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. If $d \Vdash (\eta^+, \eta^-)$, then $r_1\Downarrow_d = d \cdot r'\Downarrow_{d \otimes c'}$ and, since $d \vdash c$ by the initial assumptions, $r\Downarrow_d = d \cdot r'\Downarrow_{d \otimes c' \otimes c} = d \cdot r'\Downarrow_{d \otimes c'} = r_1\Downarrow_d$, thus the inclusion of the prefixes directly holds. Otherwise, if $d \not\Vdash (\eta^+, \eta^-)$, then the operator $\Downarrow_d$ is undefined in both cases.

$\underline{\boldsymbol{r_1 = stutt(\eta^-) \cdot r'}}\lrcorner$ By definition, $r = (c, \eta^-) \rightarrowtail c \cdot r'\downarrow_c \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. If for all $h^- \in \eta^-$, $d \nvdash h^-$, then $r_1\Downarrow_d = d$ and it holds that its prefixes are all included in the prefixes of $r \Downarrow_d = d \cdot r \Downarrow_{d \otimes c}$. Otherwise, if it exists $h^- \in \eta^-$ such that $d \vdash h^-$, then the $\Downarrow_d$ operator is undefined in both cases.

Rule R4 The conditions to apply this rule are $\langle A_1, d \rangle \not\rightarrow$, $d \vdash c$ and $d \neq \mathit{false}$, in which case the small-step behavior is defined as $\mathcal{B}^{ss}[\![D.A]\!]_d = \mathit{prefix}(d \cdot d)$. There are two cases in which it may happen that $\langle A_1, d \rangle \not\rightarrow$:

A_1 = skip By (3.1.2), $\boxtimes \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and $r = (c, \varnothing) \rightarrowtail c \cdot \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. We now have that $r \Downarrow_d = d \cdot (d \otimes c) = d \cdot d$, whose prefixes coincide with $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d$.

$A_1 = \sum_{i=1}^{n}$ ask$(c_i) \rightarrow B_i$ and $\forall 1 \leq i \leq n \nvdash c_i$ By (3.1.7), $\mathit{stutt}(\{c_1, \dots, c_n\}) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and, as a consequence, $r = (c, \{c_1, \dots, c_n\}) \rightarrowtail c \cdot r'$ belongs to $\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Now we compute $r \Downarrow_d$ and we get the trace $d \cdot r' \Downarrow_{d \otimes c} = d \cdot r' \Downarrow_d$. By definition of the evaluation function $\mathcal{A}$, r' is different from the empty conditional trace ϵ (by (3.1.7) a *stutt* construct is always followed by another conditional state). Therefore, $r' \Downarrow_d = d \cdot d \cdot s$ for some behavioral trace s. As a consequence, the behavior of the agent $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d = d \cdot d$ is included in the set of prefixes of $r \Downarrow_d = d \cdot d \cdot s$.

In case $d = \mathit{false}$ we are not allowed to apply any rule in Figure 2.2, so the small-step behavior is $\mathcal{B}^{ss}[\![D.A]\!]_{\mathit{false}} = \{\epsilon, \mathit{false}\}$. In this case, A_1 = skip since *false* is strong enough to entail any guard of a generic agent $\sum_{i=1}^{n}$ ask$(c_i) \rightarrow B_i$. As explained above, $\boxtimes \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and $r = (c, \varnothing) \rightarrowtail c \cdot \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$, thus $r \Downarrow_{\mathit{false}} = \mathit{false}$, and it is easy to note that $\mathcal{B}^{ss}[\![D \,.\, A]\!]_{\mathit{false}} \in \mathit{prefix}(\mathit{false})$.

Rule R5 This case is analogous to the case for Rule **R3** but, instead of executing the then branch (A_1), the else branch of the conditional agent (A_2) is taken, under the condition that $d \nvdash c$. More specifically, the conditions imposed for the application of the rule are $\langle A_2, d \rangle \rightarrow \langle A_2', d' \rangle$ and $d \nvdash c$, in which case $\mathcal{B}^{ss}[\![D \,.\, A]\!]_d = \mathcal{B}^{ss}[\![D \,.\, A_2]\!]_d$. By inductive hypothesis, we know that $\mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]} \Downarrow_d) \supseteq \mathcal{B}^{ss}[\![D \,.\, A_1]\!]_d$, thus also $\mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]} \Downarrow_d) \supseteq \mathcal{B}^{ss}[\![D \,.\, A]\!]_d$. In the following, we prove that $\mathit{prefix}(\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]} \Downarrow_d) \supseteq \mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]} \Downarrow_d)$ when $d \nvdash c$. We proceed by induction on the structure of a generic $r_2 \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ in order to find a conditional trace $r \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$ such that $\mathit{prefix}(r_2 \Downarrow_d) \subseteq \mathit{prefix}(r \Downarrow_d)$.

$r_2 = \boxtimes$ In this case, $r = (\mathit{true}, \{c\}) \rightarrowtail \mathit{true} \cdot \boxtimes$ belongs to $\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. We have $\boxtimes \Downarrow_d = d$, whose prefixes are included in those of $r \Downarrow_d = d \cdot d$.

$r_2 = (\eta^+, \eta^-) \rightarrowtail c' \cdot r'$ In this case, $r = (\eta^+, \eta^- \cup \{c\}) \rightarrowtail c' \cdot r' \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Let us now assume that $d \Vdash (\eta^+, \eta^-)$; then, $r_2 \Downarrow_d = d \cdot r' \Downarrow_{d \otimes c'}$. In addition, since by the initial assumptions $d \nvdash c$, $r \Downarrow_d = d \cdot r' \Downarrow_{d \otimes c'}$, the inclusion of the prefixes directly holds. Otherwise, if $d \nVdash (\eta^+, \eta^-)$, then the operator $\Downarrow_d$ is undefined in both cases.

$r_2 = \mathit{stutt}(\eta^-) \cdot r'$ By definition, $r = (\mathit{true}, \eta^- \cup \{c\}) \rightarrowtail \mathit{true} \cdot r' \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Assume that for all $h^- \in \eta^-$, $d \nvdash h^-$. Then, $r_2 \Downarrow_d = d$, and its prefixes are all included in the prefixes of $r \Downarrow_d = d \cdot r' \Downarrow_d$. Otherwise, if it exists $h^- \in \eta^-$ such that $d \vdash h^-$, then the $\Downarrow_d$ operator is undefined in both cases.

Rule R6 This case is analogous to the case for Rule **R4**. Now, the conditions to apply the rule are that $\langle A_2, d \rangle \not\rightarrow$ and $d \nvdash c$. In this case, the small-step behavior is $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d = \mathit{prefix}(d \cdot d)$. There are two cases in which it may happen that $\langle A_2, d\rangle \not\rightarrow$:

$\underline{A_2 = \mathsf{skip}}$ By (3.1.2), $\boxtimes \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ and $r = (\mathit{true}, \{c\}) \rightarrowtail \mathit{true} \cdot \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Then, since $d \not\Vdash c$, we have that $r\Downarrow_d = d \cdot d$, which coincides with $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d$.

$\underline{A_2 = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to B_i \text{ and } \forall 1 \le i \le n \not\Vdash c_i}$ By (3.1.7), $\mathit{stutt}(\{c_1, \dots, c_n\}) \cdot r' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ and, as a consequence, $r = (c, \{c_1, \dots, c_n\}) \rightarrowtail c \cdot r'$ belongs to $\mathcal{A}[\![A]\!]_{\mathcal{F}[\![D]\!]}$. Now, we compute $r\Downarrow_d$ and we get as result the trace $d \cdot r'\Downarrow_d$. Since, by definition of the semantics evaluation function $\mathcal{A}$, a *stutt* is always followed by another conditional tuple, then r' is different from the empty trace. Therefore, $r'\Downarrow_d = d \cdot s$ for some trace s. As a consequence, the behavior of the agent $\mathcal{B}^{ss}[\![D\,.\,A]\!]_d = d \cdot d$ is included in the set of prefixes of $r\Downarrow_d = d \cdot d \cdot s$.

$\underline{A = A_1 \parallel A_2}$ We prove the two directions separately.

$\subseteq$ We distinguish five different cases. Let $r := r_1 \mathbin{\bar{\parallel}} r_2 \in \mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]}$ such that $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and $r_2 \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$.

$\underline{r = r_1}$ By Definition 3.1.10, r_1 is a generic conditional trace and $r_2 = \boxtimes$ (or $r_2 = \epsilon$). In other words, r_2 is associated to an agent that adds no information. We have:

$$\begin{aligned}
\mathit{prefix}((r_1 \mathbin{\bar{\parallel}} r_2)\Downarrow_d) &= \mathit{prefix}(r_1\Downarrow_d)\\
&= \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d\\
&\qquad [\text{by Inductive Hypothesis}]\\
&\subseteq \mathcal{B}^{ss}[\![D\,.\,A_1]\!]_d\\
&= \mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_d
\end{aligned}$$

Since A_2 does not modify the store, we can conclude that the two behaviors $\mathcal{B}^{ss}[\![D\,.\,A_1]\!]_d$ and $\mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_d$ coincide.

$\underline{r = \mathit{stutt}(\eta_1^- \cup \eta_2^-) \cdot r'}$ In case $d \not\Vdash h^-\ \forall h^- \in (\eta^- \cup \delta^-)$, we have that $\mathit{prefix}(r\Downarrow_d) = \mathit{prefix}(d) = \{\epsilon,\ d\} \subseteq \mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_d$
Otherwise, $r\Downarrow_d$ is not defined, thus, the set $\mathit{prefix}(r\Downarrow_d)$ is empty and the inclusion directly holds.

$\underline{r = (\eta \otimes \delta) \rightarrowtail c_1 \otimes c_2 \cdot (r_1'{\downarrow_{c_2}} \mathbin{\bar{\parallel}} r_2'{\downarrow_{c_1}})}$ By Definition 3.1.10, $r_1 = \eta \rightarrowtail c_1 \cdot r_1' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$, $r_2 = \eta \rightarrowtail c_2 \cdot r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ and $(c_1 \otimes c_2) \neq \mathit{false}$. Let us distinguish three sub-cases.

$\underline{d \Vdash (\eta \otimes \delta) \text{ and } d \neq \mathit{false}}$ Due to the form of r_1 and r_2, we know that there exist two agents A_1' and A_2' such that $\langle A_1, d\rangle \to \langle A_1', d \otimes c_1\rangle$ and $\langle A_2, d\rangle \to \langle A_2', d \otimes c_2\rangle$, respectively. Then,

$$\begin{aligned}
&\mathit{prefix}(r\Downarrow_d) =\\
&= \mathit{prefix}(\{d \cdot (r_1'{\downarrow_{c_2}} \mathbin{\bar{\parallel}} r_2'{\downarrow_{c_1}})\Downarrow_{d \otimes c_1 \otimes c_2} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]}, r_2' \in \mathcal{A}[\![A_2']\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [c_1 \text{ and } c_2 \text{ are already in the stores of } r_1 \text{ and } r_2\text{, respectively}]
\end{aligned}$$

$$
\begin{aligned}
&= \mathit{prefix}(\{d \cdot (r'_1{\downarrow_{c_1 \otimes c_2}} \mathbin{\bar{\|}} r'_2{\downarrow_{c_1 \otimes c_2}}){\Downarrow_{d \otimes c_1 \otimes c_2}} \mid r'_1 \in \mathcal{A}\llbracket A'_1 \rrbracket_{\mathcal{F}\llbracket D \rrbracket} \\
&\qquad\qquad \text{and } r'_2 \in \mathcal{A}\llbracket A'_2 \rrbracket_{\mathcal{F}\llbracket D \rrbracket}\}) \\
&\qquad [\text{by Lemma 3.A.3}] \\
&= \mathit{prefix}(\{d \cdot (r'_1 \mathbin{\bar{\|}} r'_2){\downarrow_{c_1 \otimes c_2}}{\Downarrow_{d \otimes c_1 \otimes c_2}} \mid r'_1 \in \mathcal{A}\llbracket A'_1 \rrbracket_{\mathcal{F}\llbracket D \rrbracket} \\
&\qquad\qquad \text{and } r'_2 \in \mathcal{A}\llbracket A'_2 \rrbracket_{\mathcal{F}\llbracket D \rrbracket}\}) \\
&\qquad [\text{by Lemma 3.A.6}] \\
&= \mathit{prefix}(\{d \cdot (r'_1 \mathbin{\bar{\|}} r'_2){\Downarrow_{d \otimes c_1 \otimes c_2}} \mid r'_1 \in \mathcal{A}\llbracket A'_1 \rrbracket_{\mathcal{F}\llbracket D \rrbracket} \\
&\qquad\qquad \text{and } r'_2 \in \mathcal{A}\llbracket A'_2 \rrbracket_{\mathcal{F}\llbracket D \rrbracket}\}) \\
&\qquad [\text{by Lemma 3.A.8}] \\
&= \mathit{prefix}(\{d \cdot (r'_1{\Downarrow_{d \otimes c_1 \otimes c_2}} \mathbin{\breve{\|}} r'_2{\Downarrow_{d \otimes c_1 \otimes c_2}}) \mid r'_1 \in \mathcal{A}\llbracket A'_1 \rrbracket_{\mathcal{F}\llbracket D \rrbracket} \\
&\qquad\qquad \text{and } r'_2 \in \mathcal{A}\llbracket A'_2 \rrbracket_{\mathcal{F}\llbracket D \rrbracket}\}) \\
&\qquad [\text{by Equation (3.1.1)}] \\
&= \{\epsilon, d\} \cup \{d \cdot (s'_1 \mathbin{\breve{\|}} s'_2) \mid s'_1 \in \mathit{prefix}(\mathcal{A}\llbracket A'_1 \rrbracket_{\mathcal{F}\llbracket D \rrbracket}{\Downarrow_{d \otimes c_1 \otimes c_2}}), \\
&\qquad\qquad s'_2 \in \mathit{prefix}(\mathcal{A}\llbracket A'_2 \rrbracket_{\mathcal{F}\llbracket D \rrbracket}{\Downarrow_{d \otimes c_1 \otimes c_2}})\} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&\subseteq \{\epsilon, d\} \cup \{d \cdot (s'_1 \mathbin{\breve{\|}} s'_2) \mid s'_1 \in \mathcal{B}^{ss}\llbracket D \,.\, A'_1 \rrbracket_{d \otimes c_1 \otimes c_2},\ s'_2 \in \mathcal{B}^{ss}\llbracket D \,.\, A'_2 \rrbracket_{d \otimes c_1 \otimes c_2}\} \\
&\qquad [\text{by Definition 3.A.7 and by Definition 3.1.1}] \\
&\subseteq \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathcal{B}^{ss}\llbracket D \,.\, A'_1 \parallel A'_2 \rrbracket_{d \otimes c_1 \otimes c_2}\} \\
&\qquad [\text{by Rule } \mathbf{R7}] \\
&\subseteq \mathcal{B}^{ss}\llbracket D \,.\, A_1 \parallel A_2 \rrbracket_d
\end{aligned}
$$

$\underline{\boldsymbol{d = \mathit{false}}}\lrcorner$ We have that $\langle A_1, \mathit{false}\rangle \not\rightarrow$ and $\langle A_2, \mathit{false}\rangle \not\rightarrow$, thus $\mathcal{B}^{ss}\llbracket D \,.\, A_1 \parallel A_2 \rrbracket_{\mathit{false}} = \{\epsilon, \mathit{false}\}$. Since $d \Vdash (\eta \otimes \delta)$, we have that $\mathit{prefix}(r{\Downarrow_{\mathit{false}}}) = \{\epsilon, \mathit{false}\}$, which corresponds to the small-step behavior $\mathcal{B}^{ss}\llbracket D \,.\, A_1 \parallel A_2 \rrbracket_{\mathit{false}}$.

$\underline{\boldsymbol{d \not\Vdash (\eta \otimes \delta)}}\lrcorner$ In this case the set $\mathit{prefix}(r{\Downarrow_d})$ is empty since ${\Downarrow_d}$ is not defined under these conditions, thus it is directly included in $\mathcal{B}^{ss}\llbracket D . A_1 \parallel A_2 \rrbracket_d$.

$\underline{\boldsymbol{r = (\eta \otimes \delta) \rightarrowtail \mathit{false} \cdot \boxtimes}}\lrcorner$ By Definition 3.1.10 we have that $r_1 = \eta \rightarrowtail c_1 \cdot r'_1$, $r_2 = \delta \rightarrowtail c_2 \cdot r'_2$ and $c_1 \otimes c_2 = \mathit{false}$. We have to consider three cases:

$\underline{\boldsymbol{d \Vdash (\eta \otimes \delta) \text{ and } d \neq \mathit{false}}}\lrcorner$

$$
\begin{aligned}
\mathit{prefix}(r{\Downarrow_d}) &= \mathit{prefix}(d \cdot c_1 \otimes c_2) \\
&= \mathit{prefix}(d \cdot \mathit{false}) \\
&= \{d \cdot s \mid s \in \mathcal{B}^{ss}\llbracket D \,.\, A'_1 \parallel A'_2 \rrbracket_{\mathit{false}}\} \\
&\qquad [\text{by Rule } \mathbf{R7}] \\
&\subseteq \mathcal{B}^{ss}\llbracket D \,.\, A_1 \parallel A_2 \rrbracket_d
\end{aligned}
$$

In fact, also the second component of the behavior is the store *false*. This case represents the situation in which the contribution of the two conditional traces results in an inconsistent conditional trace.

$\underline{\boldsymbol{d = false}}\lrcorner$ We have that $\langle A_1, \mathit{false}\rangle \not\rightarrow$ and $\langle A_2, \mathit{false}\rangle \not\rightarrow$, thus $\mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_{\mathit{false}} = \{\epsilon, \mathit{false}\}$. Since $d \Vdash (\eta \otimes \delta)$, we have that $\mathit{prefix}(r{\Downarrow}_{\mathit{false}}) = \{\epsilon, \mathit{false}\}$, which corresponds to the small-step behavior $\mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_{\mathit{false}}$.

$\underline{\boldsymbol{d \nVdash (\eta \otimes \delta)}}\lrcorner$ In this case, $r{\Downarrow}_d$ is undefined, thus we have that $\varnothing \subseteq \mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_d$.

$\underline{\boldsymbol{r = (\eta^+, \eta^- \cup \delta^-) \rightarrowtail c_1 \cdot (r_1' \,\bar{\parallel}\, r_2'{\downarrow}_{c_1})}}\lrcorner$ By Definition 3.1.10, $r_1 = \eta \rightarrowtail c_1 \cdot r_1' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$, $r_2 = \mathit{stutt}(\delta^-)\cdot r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ with r_2' that recursively belongs to $\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ and for all $h^- \in \delta^-$, $\eta^+ \nvdash h^-$. Let us distinguish three subcases.

$\underline{\boldsymbol{d \Vdash (\eta^+, \eta^- \cup \delta^-)}}\lrcorner$. Then,

$$
\begin{aligned}
&\mathit{prefix}(r{\Downarrow}_d)\\
&= \mathit{prefix}(\{d\cdot(r_1' \,\bar{\parallel}\, r_2'{\downarrow}_{c_1}){\Downarrow}_{d\otimes c_1} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]},\ r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\, c_1 \text{ is already contained in the stores of } r_1 \,]\\
&= \mathit{prefix}(\{d\cdot(r_1' \,\bar{\parallel}\, r_2'){\downarrow}_{c_1}{\Downarrow}_{d\otimes c_1} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]},\ r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\,\text{by Lemma 3.A.6}\,]\\
&= \mathit{prefix}(\{d\cdot(r_1' \,\bar{\parallel}\, r_2'){\Downarrow}_{d\otimes c_1} \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]},\ r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\,\text{by Lemma 3.A.8}\,]\\
&= \mathit{prefix}(\{d\cdot(r_1'{\Downarrow}_{d\otimes c_1} \,\breve{\parallel}\, r_2'{\Downarrow}_{d\otimes c_1}) \mid r_1' \in \mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]},\ r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\,\text{by Equation (3.1.1)}\,]\\
&= \{\epsilon, d\} \cup \{d\cdot(s_1' \,\breve{\parallel}\, s_2') \mid s_1' \in \mathit{prefix}(\mathcal{A}[\![A_1']\!]_{\mathcal{F}[\![D]\!]}{\Downarrow}_{d\otimes c_1}),\\
&\qquad\qquad\qquad\qquad\qquad\quad s_2' \in \mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow}_{d\otimes c_1})\}\\
&\qquad [\,\text{by Inductive Hypothesis}\,]\\
&\subseteq \{\epsilon, d\} \cup \{d\cdot(s_1' \,\breve{\parallel}\, s_2') \mid s_1' \in \mathcal{B}^{ss}[\![D\,.\,A_1']\!]_{d\otimes c_1},\ s_2' \in \mathcal{B}^{ss}[\![D\,.\,A_2]\!]_{d\otimes c_1}\}\\
&\qquad [\,\text{by Definition 3.A.7 and by Definition 3.1.1}\,]\\
&\subseteq \{\epsilon, d\} \cup \{d\cdot s \mid s \in \mathcal{B}^{ss}[\![D\,.\,A_1' \parallel A_2]\!]_{d\otimes c_1}\}\\
&\qquad [\,\text{by Rule } \mathbf{R8}\,]\\
&\subseteq \mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_d
\end{aligned}
$$

$\underline{\boldsymbol{d = false}}\lrcorner$ We have that $\langle A_1, \mathit{false}\rangle \not\rightarrow$ and $\langle A_2, \mathit{false}\rangle \not\rightarrow$, thus $\mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_{\mathit{false}} = \{\epsilon, \mathit{false}\}$. Since $d \Vdash (\eta \otimes \delta)$, we have that $\mathit{prefix}(r{\Downarrow}_{\mathit{false}}) = \{\epsilon, \mathit{false}\}$, which corresponds to the small-step behavior $\mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_{\mathit{false}}$.

$\underline{\boldsymbol{d \nVdash (\eta^+, \eta^- \cup \delta^-)}}\lrcorner$ In this case, we have that $\mathit{prefix}(r{\Downarrow}_d) = \varnothing \subseteq \mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_d$.

$\supseteq$ We show that if $s \in \mathcal{B}^{ss}[\![D\,.\,A_1 \parallel A_2]\!]_d$, then $s \in \mathit{prefix}(\mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow}_d)$, i.e., we can find a conditional trace $r \in \mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]}$ such that $s \in \mathit{prefix}(r{\Downarrow}_d)$. We have four possible cases, depending on the rules defining the operational semantics for the agent.

1. If $\langle A_1,\ d\rangle \to \langle A_1',\ d_1'\rangle$ and $\langle A_2,\ d\rangle \to \langle A_2',\ d_2'\rangle$, the behavior of the parallel composition is $\mathcal{B}^{ss}[\![D\ .\ A_1 \parallel A_2]\!]_d = \{d\cdot s' \mid s' \in \mathcal{B}^{ss}[\![D\ .\ A_1' \parallel A_2']\!]_{d_1'\otimes d_2'}\}$. Let s be an element of that set. By inductive hypothesis, we know that there exist $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and $r_2 \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ such that $d\cdot s_1' \in \mathit{prefix}(r_1{\Downarrow_d})$ and $d\cdot s_2' \in \mathit{prefix}(r_2{\Downarrow_d})$, with $s_1' \in \mathcal{B}^{ss}[\![D\ .\ A_1']\!]_d$ and $s_2' \in \mathcal{B}^{ss}[\![D\ .\ A_2']\!]_d$. Now, consider $r = r_1 \mathbin{\bar{\parallel}} r_2$; this conditional trace belongs to $\mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]}$ whenever r_1 and r_2 are compatible via parallel composition (i.e., $r_1 \mathbin{\bar{\parallel}} r_2$ is a valid conditional trace). We show that $s \in \mathit{prefix}((r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_d})$.

$$
\begin{aligned}
&\mathit{prefix}((r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_d})\\
&\qquad\qquad [\text{by Lemma 3.A.8}]\\
&\quad = \mathit{prefix}(r_1{\Downarrow_d} \mathbin{\breve{\parallel}} r_2{\Downarrow_d})\\
&\qquad\qquad [\text{by Definition 3.1.25}]\\
&\quad = \{\epsilon, d\} \cup \{(d\cdot s_1') \mathbin{\breve{\parallel}} (d\cdot s_2') \mid s_1' \in \mathcal{B}^{ss}[\![D\ .\ A_1']\!]_{d_1'} \text{ and } s_2' \in \mathcal{B}^{ss}[\![D\ .\ A_2']\!]_{d_2'}\}\\
&\qquad\qquad [\text{by Definition 3.A.7}]\\
&\quad = \{\epsilon, d\} \cup \{d\cdot s_1' \mathbin{\breve{\parallel}} s_2' \mid s_1' \in \mathcal{B}^{ss}[\![D\ .\ A_1']\!]_{d_1'} \text{ and } s_2' \in \mathcal{B}^{ss}[\![D\ .\ A_2']\!]_{d_2'}\}\\
&\qquad\qquad [\text{by Definition 3.A.7 and by Definition 3.1.1}]\\
&\quad = \{\epsilon, d\} \cup \{d\cdot s' \mid s' \in \mathcal{B}^{ss}[\![D\ .\ A_1']\!]_{d_1'} \mathbin{\breve{\parallel}} \mathcal{B}^{ss}[\![D\ .\ A_2']\!]_{d_2'}\}\\
&\qquad\qquad [\text{by Rule } \mathbf{R7} \text{ and Equation (3.A.1)}]\\
&\quad = \{\epsilon, d\} \cup \{d\cdot s' \mid s' \in \mathcal{B}^{ss}[\![D\ .\ A_1' \parallel A_2']\!]_{d_1'\otimes d_2'}\}
\end{aligned}
$$

It follows directly that $s \in \mathit{prefix}((r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_d})$.

2. If $\langle A_1,\ d\rangle \to \langle A_1',\ d_1'\rangle$ and $\langle A_2,\ d\rangle \not\to$, then Rule **R8** is applied and we have that $\mathcal{B}^{ss}[\![D\ .\ A_1 \parallel A_2]\!]_d = \{d\cdot s' \mid s' \in \mathcal{B}^{ss}[\![D\ .\ A_1' \parallel A_2]\!]_{d_1'}\}$. Let s be an element of that set. By inductive hypothesis, we know that it exists $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ such that $d\cdot s_1' \in \mathit{prefix}(r_1{\Downarrow_d})$ with $s_1' \in \mathcal{B}^{ss}[\![D\ .\ A_1']\!]_d$. Moreover, it exists $r_2 \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}$ such that $r_2{\Downarrow_d} = d$. We distinguish two cases (corresponding to the two agents that can make the agent A_2 not to proceed) in order to prove that $s \in \mathit{prefix}((r_1 \mathbin{\bar{\parallel}} r_2){\Downarrow_d})$.

   $\underline{A_2 = \mathsf{skip}}$ In this case, the behavior of the parallel composition is that of A_1 since A_2 makes no contribution to the computation. Then, $(r_1 \mathbin{\bar{\parallel}} \boxtimes){\Downarrow_d} = d\cdot s'$ with $s' \in \mathcal{B}^{ss}[\![D\ .\ A_1']\!]_d = \mathcal{B}^{ss}[\![D\ .\ A_1' \parallel A_2]\!]_d$, thus $s \in \mathit{prefix}((r_1 \mathbin{\bar{\parallel}} \boxtimes){\Downarrow_d})$.

   $\underline{A_2 = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to B_i}$ Consider $r_2 = \mathit{stutt}(\{c_1,\dots,c_n\})\cdot r_2'$ with $r_2' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$. We can assume that $d \nVdash c_i$ for all c_i, otherwise, the agent A_2 would proceed.

$$
\begin{aligned}
&\mathit{prefix}((r_1 \mathbin{\bar{\parallel}} \mathit{stutt}(c_1,\dots,c_n)\cdot r_2'){\Downarrow_d}) =\\
&= \mathit{prefix}(\{d\cdot(r_1' \mathbin{\bar{\parallel}} r_2'){\Downarrow_{d_1'}} \mid r_1' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\text{by Lemma 3.A.8}]\\
&= \mathit{prefix}(\{d\cdot(r_1'{\Downarrow_{d_1'}} \mathbin{\breve{\parallel}} r_2'{\Downarrow_{d_1'}}) \mid r_1' \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]} \text{ and } r_2' \in \mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]}\})\\
&\qquad [\text{by Equation (3.1.1)}]
\end{aligned}
$$

$$= \{\epsilon, d\} \cup \{d \cdot (s'_1 \mathbin{\breve{\|}} s'_2) \mid s'_1 \in \mathit{prefix}(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]} \Downarrow_{d'_1}) \text{ and } s'_2 \in \mathit{prefix}(\mathcal{A}[\![A_2]\!]_{\mathcal{F}[\![D]\!]} \Downarrow_{d'_1})\}$$

[by Inductive Hypothesis]

$$= \{\epsilon, d\} \cup \{d \cdot (s'_1 \mathbin{\breve{\|}} s'_2) \mid s'_1 \in \mathcal{B}^{ss}[\![D \,.\, A'_1]\!]_{d'_1} \text{ and } s'_2 \in \mathcal{B}^{ss}[\![D \,.\, A_2]\!]_{d'_1}\}$$

[by Definition 3.A.7 and by Definition 3.1.1]

$$= \{\epsilon, d\} \cup \{d \cdot s' \mid s' \in \mathcal{B}^{ss}[\![D \,.\, A'_1]\!]_{d'_1} \mathbin{\breve{\|}} \mathcal{B}^{ss}[\![D \,.\, A_2]\!]_{d'_1}\}$$

[by Rule **R7**, Rule **R8** and Equation (3.A.1)]

$$= \{\epsilon, d\} \cup \{d \cdot s' \mid s' \in \mathcal{B}^{ss}[\![D \,.\, A'_1 \parallel A_2]\!]_{d'_1}\}$$

It follows directly that $s \in \mathit{prefix}((r_1 \mathbin{\bar{\|}} r_2)\Downarrow_d)$.

3. If $\langle A_1,\ d\rangle \not\to$ and $\langle A_2,\ d\rangle \to \langle A'_2, d'_2\rangle$, then the situation is symmetric to the previous case, thus $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d \subseteq \mathit{prefix}(\mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]} \Downarrow_d)$.
4. Finally, if $\langle A_1,\ d\rangle \not\to$ and $\langle A_2,\ d\rangle \not\to$, then we can reason similarly to Point 2, considering, for both A_1 and A_2, the two cases in which they cannot proceed. We can conclude that $\mathcal{B}^{ss}[\![D \,.\, A_1 \parallel A_2]\!]_d = \{\epsilon,\ d\} \subseteq \mathit{prefix}(\mathcal{A}[\![A_1 \parallel A_2]\!]_{\mathcal{F}[\![D]\!]} \Downarrow_d)$.

$\underline{\boldsymbol{A = \exists x\, A_1}}$ We prove the two directions independently.

$\subseteq$ We show that: $\mathit{prefix}(\mathcal{A}[\![\exists x\, A_1]\!]_{\mathcal{F}[\![D]\!]} \Downarrow_d) \subseteq \mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_d$. Let $r = \bar{\exists}_x r_1$ such that $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ and r_1 is x-self-sufficient. We show that the prefixes of $(\bar{\exists}_x r_1)\Downarrow_d$ are included in the behavior $\mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_d$ by structural induction on r_1:

$\underline{\boldsymbol{r_1 = \epsilon}}$ The statement directly holds.

$\underline{\boldsymbol{r_1 = \boxtimes}}$ Then, $\boxtimes\Downarrow_d = d$, which belongs to $\mathcal{B}^{ss}[\![D \,.\, \exists x\, A_1]\!]_d$.

$\underline{\boldsymbol{r_1 = \eta \rightarrowtail l \cdot r'_1}}$ By Definition 3.1.16, we have that $r'_1 \in \mathcal{A}[\![A'_1]\!]_{\mathcal{F}[\![D]\!]}$ and, by inductive hypothesis, there exists a transition $\langle A_1, d\rangle \to \langle A'_1, d'\rangle$.
Since r_1 is x-self-sufficient, also r'_1 is x-self-sufficient. Now, we have three cases.

$\underline{\boldsymbol{d \Vdash \exists_x \eta \text{ and } d \neq \mathit{false}}}$

$$\mathit{prefix}(r\Downarrow_d)$$

$$= \mathit{prefix}(\{\bar{\exists}_x(\eta \rightarrowtail l \cdot r'_1)\Downarrow_d \mid r'_1 \in \mathcal{A}[\![A'_1]\!]_{\mathcal{F}[\![D]\!]} \text{ and } r'_1\ x\text{-self-sufficient}\})$$

[by Definition 3.1.25]

$$= \mathit{prefix}(\{d \cdot (\bar{\exists}_x r'_1)\Downarrow_{d \otimes \exists_x l} \mid r'_1 \in \mathcal{A}[\![A'_1]\!]_{\mathcal{F}[\![D]\!]} \text{ and } r'_1\ x\text{-self-sufficient}\})$$

[$r'_1 \in \mathcal{A}[\![A'_1]\!]_{\mathcal{F}[\![D]\!]}$ and r'_1 x-self-sufficient]

$$= \mathit{prefix}(\{d \cdot s \mid s \in (\mathcal{A}[\![\exists x\, A'_1]\!]_{\mathcal{F}[\![D]\!]})\Downarrow_{d \otimes \exists_x l}\})$$

[by Equation (3.1.1)]

$$= \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathit{prefix}(\mathcal{A}[\![\exists x\, A'_1]\!]_{\mathcal{F}[\![D]\!]} \Downarrow_{d \otimes \exists_x l})\}$$

[by Inductive Hypothesis]

$$\subseteq \{\epsilon, d\} \cup \{d \cdot s \mid s \in \mathcal{B}^{ss}[\![D \,.\, \exists x\, A'_1]\!]_{d \otimes \exists_x l}\}$$

[by Rule **R9**]

$$\subseteq \mathcal{B}^{ss}[\![D\,.\,\exists x\, A_1]\!]_d$$

$\underline{\boldsymbol{d = false}}\lrcorner$ We have that $\langle \exists x\, A_1, \mathit{false}\rangle \not\to$, thus $\mathcal{B}^{ss}[\![D.\exists x\, A_1]\!]_{\mathit{false}} = \{\epsilon, \mathit{false}\}$. On the other hand, since $d \Vdash \exists_x \eta$, we have that $\mathit{prefix}(r{\Downarrow_{\mathit{false}}}) = \{\epsilon, \mathit{false}\}$ which corresponds to the small-step behavior $\mathcal{B}^{ss}[\![D.\exists x\, A_1]\!]_{\mathit{false}}$.

$\underline{\boldsymbol{d \nVdash \exists_x \eta}}\lrcorner$ Then, the operator ${\Downarrow_d}$ is undefined for the conditional trace, thus $\mathit{prefix}(r{\Downarrow_d}) = \varnothing \subseteq \mathcal{B}^{ss}[\![D\,.\,\exists x\, A_1]\!]_d$.

$\underline{\boldsymbol{r_1 = stutt(\{c_1, \dots, c_n\}) \cdot r'_1}}\lrcorner$ By Definition 3.1.16, $r'_1 \in \mathcal{A}[\![\sum_{i=1}^{i} \mathsf{ask}(n_i) \to B_i]\!]_{\mathcal{F}[\![D]\!]}$. If it exists no index $1 \le j \le n$ such that $d \vdash c_j$, then this implies that $d \vdash \exists_x c_j$. In such case, we have

$$\begin{aligned}
\mathit{prefix}(r{\Downarrow_d}) &= \mathit{prefix}(\bar{\exists}_x(\mathit{stutt}(\{c_1, \dots, c_n\}) \cdot r'_1){\Downarrow_d}) \\
&= \mathit{prefix}(\mathit{stutt}(\{\exists_x c_1, \dots, \exists_x c_n\}) \cdot \bar{\exists}_x r'_1{\Downarrow_d}) \\
&\qquad [\text{by Definition 3.1.25}] \\
&= d \subseteq \mathcal{B}^{ss}[\![D\,.\,\exists x\, A_1]\!]_d
\end{aligned}$$

Otherwise, if it exists an index j such that $d \vdash c_j$, then $r{\Downarrow_d}$ is undefined, thus $\mathit{prefix}(r{\Downarrow_d}) = \varnothing \subseteq \mathcal{B}^{ss}[\![D\,.\,\exists x\, A_1]\!]_d$.

$\supseteq$ From Rule **R9**, we know that, if $d \neq \mathit{false}$, then $\mathcal{B}^{ss}[\![D.A_1]\!]_{l \otimes \exists_x d} = l' {\cdot} \mathcal{B}^{ss}[\![D.A'_1]\!]_d$, where l and l' are local stores. Moreover, $l = \mathit{true}$ because it is the initial (local) store for A_1. In the following, we show that $d \cdot \mathcal{B}^{ss}[\![D\,.\,\exists x\, A'_1]\!]_{d \otimes \exists_x l} \in \mathcal{A}[\![\exists x\, A_1]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_d}$, i.e., it exists a trace $r \in \mathcal{A}[\![\exists x\, A_1]\!]_{\mathcal{F}[\![D]\!]}$ such that $r{\Downarrow_d} = d \cdot s$ with $s \in \mathcal{B}^{ss}[\![D\,.\,\exists x\, A'_1]\!]_{d \otimes \exists_x l}$. By inductive hypothesis, $\mathcal{B}^{ss}[\![D\,.\,A_1]\!]_{\exists_x d} \subseteq \mathit{prefix}(\mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_{\exists_x d}})$, and by Rule **R9**, it holds that there exists $r_1 \in \mathcal{A}[\![A_1]\!]_{\mathcal{F}[\![D]\!]}$ such that $r_1{\Downarrow_{\exists_x d}} = \exists_x d \cdot \mathcal{B}^{ss}[\![D\,.\,\exists x\, A'_1]\!]_{l'}$.

Now, r_1 is x-self-sufficient since the only external information is provided by $\exists_x d$, which in fact does not contain information about x. Moreover, r_1 is of the form $\eta \rightarrowtail l' \cdot r'_1$ with $r'_1 \in \mathcal{A}[\![A'_1]\!]_{\mathcal{F}[\![D]\!]}$. Therefore, it exists $r \in \mathcal{A}[\![\exists x\, A_1]\!]_{\mathcal{F}[\![D]\!]}$ such that $r = \bar{\exists}_x r_1$. Then,

$$\begin{aligned}
r{\Downarrow_d} &= (\bar{\exists}_x \eta \rightarrowtail l' \cdot r'_1){\Downarrow_d} \\
&= (\exists_x \eta \rightarrowtail \exists_x l' \cdot \bar{\exists}_x r'_1){\Downarrow_d} \\
&\qquad [\text{by Definition 3.1.25}] \\
&= d \cdot (\bar{\exists}_x r'_1){\Downarrow_{\exists_x l' \otimes d}} \\
&\qquad [\text{by Definition 3.1.25}] \\
&= d \cdot s \qquad \text{with } s \in \mathcal{A}[\![\exists x\, A'_1]\!]_{\mathcal{F}[\![D]\!]}{\Downarrow_{\exists_x l' \otimes d}} \\
&\qquad [\text{by Inductive Hypothesis}] \\
&= d \cdot s \qquad \text{with } s \in \mathcal{B}^{ss}[\![D\,.\,\exists x\, A'_1]\!]_{\exists_x l' \otimes d}
\end{aligned}$$

If $d = \mathit{false}$, then we have that $\mathit{prefix}(r{\Downarrow_{\mathit{false}}}) = \{\epsilon, \mathit{false}\}$, which corresponds to the small-step behavior $\mathcal{B}^{ss}[\![D\,.\,\exists x\, A_1]\!]_{\mathit{false}}$ since the transition relation $\to$ is not defined for $\langle \exists x\, A_1, \mathit{false}\rangle$.

$\underline{\boldsymbol{A = p(\vec{x})}}\lrcorner$ We have to distinguish two sub-cases.

d* ≠ *false

$$\begin{aligned}
&\mathit{prefix}(\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_d)\\
&= \mathit{prefix}(\{(\mathit{true},\varnothing) \rightarrowtail \mathit{true}\cdot r' \mid r' \in \mathcal{F}[\![D]\!](p(\vec{x}))\}\Downarrow_d)\\
&\qquad [\text{since } \mathcal{F}[\![D]\!] = \mathcal{D}[\![D]\!]_{\mathcal{F}[\![D]\!]}\,]\\
&= \mathit{prefix}(\{(\mathit{true},\varnothing) \rightarrowtail \mathit{true}\cdot r' \mid r' \in \mathcal{D}[\![D]\!]_{\mathcal{F}[\![D]\!]}(p(\vec{x}))\}\Downarrow_d)\\
&\qquad [\text{by Definition 3.1.20}\,]\\
&= \mathit{prefix}(\{(\mathit{true},\varnothing) \rightarrowtail \mathit{true}\cdot r' \mid r' \in \mathcal{A}[\![B]\!]_{\mathcal{F}[\![D]\!]},\ p(\vec{x}) \mathrel{:-} B \in D\}\Downarrow_d)\\
&= \mathit{prefix}(\{d\cdot s' \mid s' \in (\mathcal{A}[\![B]\!]_{\mathcal{F}[\![D]\!]})\Downarrow_d,\ p(\vec{x}) \mathrel{:-} B \in D\})\\
&\qquad [\text{by Inductive Hypothesis}\,]\\
&= \mathit{prefix}(\{d\cdot s' \mid s' \in \mathcal{B}^{ss}[\![D\,.\,B]\!]_d,\ p(\vec{x}) \mathrel{:-} B \in D\})\\
&\qquad [\text{by Rule } \mathbf{R10}\,]\\
&= \mathcal{B}^{ss}[\![D\,.\,p(\vec{x})]\!]_d
\end{aligned}$$

Notice that, in the second last equality, the structural induction hypothesis cannot be applied because B can be structurally greater than $p(\vec{x})$. For this reason, we have to introduce a second induction on the number of $p(\vec{x})$ present on B. If B does not contain any process call $p(\vec{x})$, then we can directly apply structural induction. Otherwise, if the agent contains one process call $p(\vec{x})$, it is sufficient to replace the call with the body of the declaration. In this way, B has less process calls $p(\vec{x})$ than A and we can apply the inductive hypothesis.

d* = *false In this case, the transition relation $\rightarrow$ is not defined for the configuration $\langle p(\vec{x}), \mathit{false}\rangle$, hence

$$\begin{aligned}
&\mathit{prefix}(\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{F}[\![D]\!]}\Downarrow_{\mathit{false}})\\
&= \mathit{prefix}(\{((\mathit{true},\varnothing) \rightarrowtail \mathit{true}\cdot r')\Downarrow_{\mathit{false}} \mid r' \in \mathcal{F}[\![D]\!](p(\vec{x}))\})\\
&= \{\epsilon, \mathit{false}\}\\
&= \mathcal{B}^{ss}[\![D\,.\,p(\vec{x})]\!]_{\mathit{false}}
\end{aligned}$$

Theorem 3.1.27. *Let P_1, P_2 be two programs. Then $\mathcal{P}[\![P_1]\!] = \mathcal{P}[\![P_2]\!] \iff \forall c \in \mathbf{C}.\, \mathcal{B}^{ss}[\![P_1]\!]_c = \mathcal{B}^{ss}[\![P_2]\!]_c$.*

Proof.

By Theorem 3.1.26 it follows that for each program P and each $c \in \mathbf{C}$, $\mathit{prefix}(\mathcal{P}[\![P]\!]\Downarrow_c) = \mathcal{B}^{ss}[\![P]\!]_c$. Thus, we show that $\mathcal{P}[\![P_1]\!] = \mathcal{P}[\![P_2]\!] \iff \forall c \in \mathbf{C}.\, \mathit{prefix}(\mathcal{P}[\![P_1]\!]\Downarrow_c) = \mathit{prefix}(\mathcal{P}[\![P_2]\!]\Downarrow_c)$.

$\Rightarrow$ Follows directly from Definition 3.1.25 and by definition of *prefix*.

$\Leftarrow$ To prove this implication we first need to show that $\mathcal{P}[\![P_1]\!] \neq \mathcal{P}[\![P_2]\!] \Rightarrow \exists \bar{c} \in \mathbf{C}.\ \mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$. Without loss of generality, assume that $\mathcal{P}[\![P_1]\!] \supset \mathcal{P}[\![P_2]\!]$, thus, it exists $r_1 \in \mathcal{P}[\![P_1]\!]$ such that $r_1 \notin \mathcal{P}[\![P_2]\!]$. We can distinguish two cases: $\mathcal{P}[\![P_2]\!]$ is empty or $\mathcal{P}[\![P_2]\!]$ contains at least one conditional trace.

If $\mathcal{P}[\![P_2]\!] = \varnothing$, then $\mathcal{P}[\![P_2]\!]\Downarrow_c$ is empty for any possible $c \in \mathbf{C}$. Now, if we choose $\bar{c}$ to be the *lub* ($\otimes$) of all the positive conditions occurring in r_1, then $r_1\Downarrow_{\bar{c}}$ is a valid trace. Therefore, $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \supseteq \{r_1\Downarrow_{\bar{c}}\} \neq \varnothing$.

If $\mathcal{P}[\![P_2]\!] \neq \varnothing$, by the initial assumptions, it exists a conditional trace $r_2 \in \mathcal{P}[\![P_2]\!]$ such that $r_1 \neq r_2$. Without loss of generality, assume that $length(r_1) \leq length(r_2)$ and that r_1 differs from r_2 at position k, with $k \in [1, length(r_1)]$. The index k is guaranteed to exist.[6] We consider the six possible cases, corresponding to the possible forms of the conditional state at position k, in order to prove that there exists a store $\bar{c}$ such that $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$. In the following, the stores $\bar{c}_1$ and $\bar{c}_2$ correspond to the *lub* ($\otimes$) of all the positive conditions occurring in r_1 and r_2, respectively.

1. Let be $(\eta_1^+, \eta_1^-) \rightarrowtail d_1$ and $(\eta_2^+, \eta_2^-) \rightarrowtail d_2$ the k-th conditional tuple in r_1 and r_2, respectively. There are three possible ways in which these two tuples can differ:

   $\underline{\eta_1^+ \neq \eta_2^+}$: Let us assume that $\eta_1^+ \vdash \eta_2^+$ and $\eta_2^+ \nvdash \eta_1^+$. Notice that r_1 has to come from the semantics of an ask or a now construct since they are the only *tccp* agents that can add information to the positive condition (see Definition 3.1.16). Hence, there exists also a conditional trace $\bar{r}_1 \in \mathcal{P}[\![P_1]\!]$ in which η_1^+ occurs in a negative condition (corresponding to the else branch of a now agent) or in a *stutt* construct (corresponding to the suspension of an ask agent) of the sequence. There are two cases in which $\bar{r}_1$ does not exists, but both are in contradiction with the hypothesis: (1) when $\eta_1^+ = true$, but this contradicts $\eta_2^+ \nvdash \eta_1^+$ or (2) when a constraint d stronger than η_1^+ ($d \vdash \eta_1^+$) is propagated. In this last case, the trace $\bar{r}_1$ does not exists since the condition is in contradiction with the propagated store. However, since $\eta_1^+ \vdash \eta_2^+$, it follows that d entails also η_2^+ ($d \otimes \eta_1^+ = d \otimes \eta_2^+ = d$). Therefore, the propagation of d makes r_1 and r_2 equal. Since they were supposed to be different only at this point, this is a contradiction with the hypothesis $r_1 \neq r_2$. Therefore, $\bar{r}_1$ exists and belongs to $\mathcal{P}[\![P_1]\!]$. Furthermore, $\bar{r}_1$ differs from any trace in $\mathcal{P}[\![P_2]\!]$ for at least the negative part of a condition or the body of a *stutt*, otherwise, reasoning in a similar way as above, r_1 would also belong to $\mathcal{P}[\![P_2]\!]$, and this is not possible.
   If $\eta_1^+ \nvdash \eta_2^+$ and $\eta_2^+ \vdash \eta_1^+$, we can reason in a symmetric way, thus concluding that it exists $\bar{r}_2 \in \mathcal{P}[\![P_2]\!]$ that differs from any trace in $\mathcal{P}[\![P_1]\!]$ for at least the negative part of a condition or the body of a *stutt*.
   Finally, if $\eta_1^+ \nvdash \eta_2^+$ and $\eta_2^+ \nvdash \eta_1^+$, we can reason as before and deduce that there exist two traces $\bar{r}_1 \in \mathcal{P}[\![P_1]\!]$ and $\bar{r}_2 \in \mathcal{P}[\![P_2]\!]$, which contains respectively η_1^+ and η_2^+ in the negative part of the condition, and such that $\bar{r}_1 \notin \mathcal{P}[\![P_2]\!]$ and $\bar{r}_2 \notin \mathcal{P}[\![P_1]\!]$.
   In case $\bar{r}_1$ (respectively $\bar{r}_2$) comes from an ask agent we remand to the following Points 2, 3 and 4 of the proof, where we deal with the conditional traces containing *stutt* constructs. Otherwise, if r_1 comes from a now agent we can reduce to the following case where we deal with the negative part of the conditions ($\eta_1^- \neq \eta_2^-$).

   $\underline{\eta_1^- \neq \eta_2^-}$: Let us first assume that $\eta_1^- \subset \eta_2^-$. This means that the store at position k in r_2 has to satisfy a stronger condition than the one in r_1. Let $\bar{c} := \bar{c}_1 \otimes h_2^-$, with $h_2^- \in \eta_2^- \smallsetminus \eta_1^-$. Under these conditions, $r_1\Downarrow_{\bar{c}}$ computes a behavioral timed trace whereas $r_2\Downarrow_{\bar{c}}$ computes no trace since, at position k, $\bar{c}$ entails one of

[6] There are two cases in which k does not exist, but both are in contradiction with the initial hypothesis: (1) $r_1 = r_2$ or (2) one of the traces is a prefix of the other.

the stores in the negative condition.
For the case in which $\eta_2^- \subset \eta_1^-$ we choose $c = \bar{c}_2 \otimes h_1^-$, with $h_1^- \in \eta_1^- \smallsetminus \eta_2^-$ and reason in an symmetric way.
Finally, if $\eta_1^- \not\subseteq \eta_2^-$ and $\eta_2^- \not\subseteq \eta_1^-$, we can choose indifferently $\bar{c} = \bar{c}_1 \otimes h_2^-$ or $\bar{c} = \bar{c}_2 \otimes h_1^-$ and conclude that $r_1{\Downarrow_{\bar{c}}}$ computes a behavioral timed trace but $r_2{\Downarrow_{\bar{c}}}$ is not defined, or vice-versa.
Thus, we can conclude that $\mathcal{P}[\![P_1]\!]{\Downarrow_{\bar{c}}} \neq \mathcal{P}[\![P_2]\!]{\Downarrow_{\bar{c}}}$.

$\underline{\boldsymbol{d_1 \neq d_2}}\lrcorner$ Consider $\bar{c} = \bar{c}_1 = \bar{c}_2$. There are two possible cases. Assume first that $\bar{c} \nVdash d_1$ and $\bar{c} \nVdash d_2$. Both r_1 and r_2 must be *compatible* with their own conditions, thus, being the store monotonic, it happens that $r_1{\Downarrow_{\bar{c}}}$ and $r_2{\Downarrow_{\bar{c}}}$ are both defined. Moreover, we know that $\eta_1^+ = \eta_2^+$ and from Property 3.A.1 $d_1 \vdash \eta_1^+$ and $d_2 \vdash \eta_1^+$. Since $\bar{c} \nVdash d_1$ and $\bar{c} \nVdash d_2$, we can conclude that in $r_1{\Downarrow_{\bar{c}}}$ at position k we have the store d_1, whereas in $r_2{\Downarrow_{\bar{c}}}$ at the same position we find the store d_2 that is different from d_1 by the initial assumptions. Thus $r_1{\Downarrow_{\bar{c}}} \neq r_2{\Downarrow_{\bar{c}}}$. Assume now that $\bar{c}$ contains more information than the store d_1 (respectively d_2). Then, we know that, at certain point in r_1 (respectively r_2), the positive condition is stronger than d_1 (respectively d_2). Therefore, we can reason as in the previous case when $\eta_1^+ \neq \eta_2^+$ and r_1 (respectively r_2) are produced by the semantics of an ask or a now agent.

2. Let $\mathit{stutt}(\eta_1^-)$ (respectively $\mathit{stutt}(\eta_2^-)$) be the k-th conditional state in r_1 (respectively r_2). It is sufficient to proceed as in Point 1 of this proof (case $\eta_1^- \neq \eta_2^-$) to show that there exists a store $\bar{c}$ such that $r_1{\Downarrow_{\bar{c}}}$ is well defined while $r_2{\Downarrow_{\bar{c}}}$ is not. For instance, if $\eta_1^- \subset \eta_2^-$ we set $\bar{c} = \bar{c}_1 \otimes h_2^-$, with $h_2^- \in \eta_2^- \smallsetminus \eta_1^-$. It is easy to notice that $r_1{\Downarrow_{\bar{c}}}$ computes a behavioral timed trace but $r_2{\Downarrow_{\bar{c}}}$ recovers no trace since at position k the constraint h_2^- belongs to the negative part of the condition. Therefore, $\mathcal{P}[\![P_1]\!]{\Downarrow_{\bar{c}}} \neq \mathcal{P}[\![P_2]\!]{\Downarrow_{\bar{c}}}$.
3. Let $\eta_1 \rightarrowtail d_1$ be the k-th conditional tuple in r_1 and $\mathit{stutt}(\eta_2^-)$ the k-th element in r_2. Consider $\bar{c} = \bar{c}_1$. Up to instant k, $r_1{\Downarrow_{\bar{c}}}$ and $r_2{\Downarrow_{\bar{c}}}$ coincide and, as r_1 and r_2 differ only at position k, $\bar{c}$ satisfies all the conditions in r_1 and in r_2 till up that position. The behavioral timed trace $r_2{\Downarrow_{\bar{c}}}$ ends at position k since a *stutt* has been encountered (see Definition 3.1.25). However, since r_1 is maximal, $r_1{\Downarrow_{\bar{c}}}$ does not end at position k but continues with at least another state, otherwise we would have found an ending symbol $\boxtimes$. In conclusion, $r_2{\Downarrow_{\bar{c}}}$ is at least one store longer than $r_1{\Downarrow_{\bar{c}}}$, thus, $\mathcal{P}[\![P_1]\!]{\Downarrow_{\bar{c}}} \neq \mathcal{P}[\![P_2]\!]{\Downarrow_{\bar{c}}}$.
4. If $\eta_2 \rightarrowtail d_2$ is the k-th element in r_2 and $\mathit{stutt}(\eta_1^-)$ that in r_1, then the proof is symmetric to previous Point 3.
5. Let $\boxtimes$ and $\eta_2 \rightarrowtail d_2$ be the k-th states of r_1 and r_2, respectively. We can reason similarly to Point 3 above in this proof by choosing $\bar{c} = \bar{c}_2$. By hypothesis, r_1 and r_2 differ only at position k, thus, $r_1{\Downarrow_{\bar{c}}}$ and $r_2{\Downarrow_{\bar{c}}}$ compute the same behavioral timed trace up to position k-th. However, while $r_1{\Downarrow_{\bar{c}}}$ stops at instant k (an ending symbol $\boxtimes$ is found), $r_2{\Downarrow_{\bar{c}}}$ is at least one store longer. Thus, $\mathcal{P}[\![P_1]\!]{\Downarrow_{\bar{c}}} \neq \mathcal{P}[\![P_2]\!]{\Downarrow_{\bar{c}}}$.
6. Let $\boxtimes$ be the k-th element of r_1 and $\mathit{stutt}(\eta_2)$ the conditional state occurring in r_2 at the same position. We set $\bar{c} = \bar{c}_1 \otimes h_2^-$, with $h_2^- \in \eta_2^- \smallsetminus \eta_1^-$. In this way, $r_1{\Downarrow_{\bar{c}}}$ is defined but $r_2{\Downarrow_{\bar{c}}}$ computes no trace since, at position k, the constraint h_2^- is required not to be entailed by the current store. Thus, $\mathcal{P}[\![P_1]\!]{\Downarrow_{\bar{c}}} \neq \mathcal{P}[\![P_2]\!]{\Downarrow_{\bar{c}}}$.

In conclusion, we can always choose an adequate $\bar{c}$ which differentiates $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}}$ from $\mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$. From Definition 3.1.16 and Definition 3.1.20, it can be noticed that the traces contained in $\mathcal{P}[\![P_1]\!]$ and $\mathcal{P}[\![P_2]\!]$ either end in $\boxtimes$ or are infinite. From this observation, it follows directly that, if $\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}} \neq \mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}}$, then $\mathit{prefix}(\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}}) \neq \mathit{prefix}(\mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}})$. Otherwise, there would exists a trace in $\mathcal{P}[\![P_1]\!]$ that is prefix of a trace in $\mathcal{P}[\![P_2]\!]$ (or viceversa), which is not possible since $\boxtimes$ is a termination symbol and an infinite trace cannot prefix another infinite trace. Thus, we can conclude that if $\mathcal{P}[\![P_1]\!] \neq \mathcal{P}[\![P_2]\!]$, then there exists $\bar{c} \in \mathbf{C}$ such that $\mathit{prefix}(\mathcal{P}[\![P_1]\!]\Downarrow_{\bar{c}}) \neq \mathit{prefix}(\mathcal{P}[\![P_2]\!]\Downarrow_{\bar{c}})$, and this concludes the proof.

Proposition 3.1.28. *Let $D_1, D_2 \in \mathbf{D}^{\Pi}_{\mathbf{C}}$. Then $D_1 \approx_{\mathcal{F}} D_2 \iff \forall A \in \mathbf{A}^{\Pi}_{\mathbf{C}}. \mathcal{P}[\![D_1 \,.\, A]\!] = \mathcal{P}[\![D_2 \,.\, A]\!]$.*

Proof.

$\Rightarrow$ Straightforward.

$\Leftarrow$ By Definition 3.1.20, $\mathcal{P}[\![D_1 \,.\, A]\!] = \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D_1]\!]}$ and $\mathcal{P}[\![D_2 \,.\, A]\!] = \mathcal{A}[\![A]\!]_{\mathcal{F}[\![D_2]\!]}$. We have to check that $\mathcal{F}[\![D_1]\!] = \mathcal{F}[\![D_2]\!]$. The only case depending on the interpretation is when $A = p(\vec{x})$. By hypothesis,

$$\begin{aligned}\mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{F}[\![D_1]\!]} &= \bigsqcup\{(\mathit{true},\varnothing) \rightarrowtail \mathit{true}\cdot r \mid r \in \mathcal{F}[\![D_1]\!](p(\vec{x}))\}\\ &= \bigsqcup\{(\mathit{true},\varnothing) \rightarrowtail \mathit{true}\cdot r \mid r \in \mathcal{F}[\![D_2]\!](p(\vec{x}))\} = \mathcal{A}[\![p(\vec{x})]\!]_{\mathcal{F}[\![D_2]\!]}\end{aligned}$$

We have to check that $\mathcal{F}[\![D_1]\!](p(\vec{x}))$ and $\mathcal{F}[\![D_2]\!](p(\vec{x}))$ coincide for each $p(\vec{x}) \in \mathbf{PC}$. Since $\mathcal{F}[\![D_1]\!]$ (respectively $\mathcal{F}[\![D_2]\!]$) is the least fixpoint of $\mathcal{D}[\![D_1]\!]_\perp$ (respectively $\mathcal{D}[\![D_2]\!]_\perp$), we know that it contains only information regarding the procedure calls in D_1 (respectively D_2). So we can conclude that $\mathcal{F}[\![D_1]\!] = \mathcal{F}[\![D_2]\!]$.

Corollary 3.1.29. *Let $D_1, D_2 \in \mathbf{D}^{\Pi}_{\mathbf{C}}$. Then $D_1 \approx_{ss} D_2$ if and only if $D_1 \approx_{\mathcal{F}} D_2$.*

Proof.

Consider $D_1, D_2 \in \mathbf{D}^{\Pi}_{\mathbf{C}}$:

$$\begin{aligned}D_1 \approx_{\mathcal{F}} D_2 &\Leftrightarrow \mathcal{F}[\![D_1]\!] = \mathcal{F}[\![D_2]\!]\\ &\qquad [\text{by Proposition 3.1.28}]\\ &\Leftrightarrow \forall A \in \mathbf{A}^{\Pi}_{\mathbf{C}}. \mathcal{P}[\![D_1 \,.\, A]\!] = \mathcal{P}[\![D_2 \,.\, A]\!]\\ &\qquad [\text{by Theorem 3.1.27}]\\ &\Leftrightarrow \forall A \in \mathbf{A}^{\Pi}_{\mathbf{C}} \forall c \in \mathbf{C}. \mathit{prefix}(\mathcal{P}[\![D_1 \,.\, A]\!]\Downarrow_c) = \mathit{prefix}(\mathcal{P}[\![D_2 \,.\, A]\!]\Downarrow_c)\\ &\qquad [\text{by Theorem 3.1.26}]\\ &\Leftrightarrow \forall A \in \mathbf{A}^{\Pi}_{\mathbf{C}} \forall c \in \mathbf{C}. \mathcal{B}^{ss}[\![D_1 \,.\, A]\!]_c = \mathcal{B}^{ss}[\![D_2 \,.\, A]\!]_c\\ &\Leftrightarrow D_1 \approx_{ss} D_2\end{aligned}$$

3.A.2 Proofs of Section 3.2

Lemma 3.A.9 $(\mathbf{M}, \sqsubseteq, \sqcup, \sqcap, \mathbf{M}, \{\epsilon\}) \underset{\alpha_{io}}{\overset{\gamma_{io}}{\leftrightarrows}} (\mathbf{IO}, \subseteq, \cup, \cap, \mathbf{IO}, \varnothing)$

Proof.

α_{io} is monotonic Let $R_1, R_2 \in \mathbf{M}$ such that $R_1 \sqsubseteq R_2$, thus, $\alpha_{io}(R_1) \subseteq \alpha_{io}(R_2)$. Otherwise, if there exists an input-output pair belonging to $\alpha_{io}(R_1)$ but not to $\alpha_{io}(R_2)$, this means that the associated trace belongs to R_1 but not to R_2, and this contradicts the hypothesis.

γ_{io} is monotonic Let $P_1, P_2 \in \mathbf{IO}$ such that $P_1 \subseteq P_2$. Suppose that $\gamma_{io}(P_1) \not\sqsubseteq \gamma_{io}(P_2)$, in this case, there exists $r_1 \in \gamma_{io}(P_1)$ but not $r_2 \in \gamma_{io}(P_1)$ that extends r_1 (r_1 is a prefix of r_2). It is easy to see that this situation is impossible since, by the definition of γ_{io}, r_1 has to belong also to $\gamma_{io}(P_2)$ (since $P_1 \subseteq P_2$) and r_1 trivially extends itself.

$(\gamma_{io} \circ \alpha_{io})$ is extensive This means that for all $R \in \mathbf{M}$, $R \sqsubseteq \gamma_{io}(\alpha_{io}(R))$. We show that $r \in R \Rightarrow r \in \gamma_{io}(\alpha_{io}(R))$; we distinguish three cases:

$r = \eta_1 \rightarrowtail c_1 \cdot \cdots \cdot \eta_n \rightarrowtail c_n \cdot \boxtimes$ We have that:

$$\alpha_{io}(R) \supseteq \{\langle c_0, \mathit{fin}(c)\rangle \mid c_0 \in \mathbf{C} \text{ and } \mathit{last}(r{\Downarrow}_{c_0}) = c\}.$$

Thus, by (3.2.2), it follows that $r \in \gamma_{io}(\alpha_{io}(r))$.

$r = \eta_1 \rightarrowtail c_1 \cdot \cdots \cdot \mathit{stutt}(\eta_n^-) \cdot \ldots$ We have that:

$$\alpha_{io}(R) \supseteq \{\langle c_0, \mathit{fin}(c)\rangle \mid c_0 \in \mathbf{C} \text{ and } \mathit{last}(r{\Downarrow}_{c_0}) = c\}.$$

From (3.2.2), it follows that $r \in \gamma_{io}(\alpha_{io}(r))$.

$r = \eta_1 \rightarrowtail c_1 \ldots \eta_n \rightarrowtail c_n \cdots$ (an infinite sequence that does not contain any *stutt*). We have that $\alpha_{io}(R) \supseteq \{\langle c_0, \mathit{inf}(c)\rangle \mid c_0 \in \mathbf{C},\ r{\Downarrow}_{c_0} = c'_0 \ldots c'_i \ldots, \text{ and } \otimes_{i \geq 0} c'_i = c\}$. By (3.2.2), we have that $r \in \gamma_{io}(\alpha_{io}(r))$.

$(\alpha_{io} \circ \gamma_{io})$ is the identity for IO This means that for all $P \in \mathbf{IO}$, $P = \alpha_{io}(\gamma_{io}(P))$. We show the two inclusions separately.

$\subseteq$ We first show that $p \in P \Rightarrow p \in \alpha_{io}(\gamma_{io}(P))$ by distinguishing two sub-cases.

$p = \langle c_0, \mathit{fin}(c_n)\rangle$ In this case, $\gamma_{io}(P)$ contains all the conditional traces r such that $\mathit{last}(r{\Downarrow}_{c_0}) = c_n$. By (3.2.1), $p \in \alpha_{io}(\gamma_{io}(P))$.

$p = \langle c_0, \mathit{inf}(c)\rangle$ We have that $\gamma_{io}(P)$ contains all the conditional state sequences r such that $r{\Downarrow}_{c_0} = c_0 \ldots c_i \ldots$ and $\otimes_{i \geq 0} = c$. By (3.2.1), $p \in \alpha_{io}(\gamma_{io}(P))$.

$\supseteq$ Now we show the other inclusion i.e., $p \in \alpha_{io}(\gamma_{io}(P)) \Rightarrow p \in P$. We have to consider two sub-cases.

$p = \langle c_0, \mathit{fin}(c_n)\rangle$ In this case, it exists $r \in \gamma_{io}(P)$ such that $\mathit{last}(r{\Downarrow}_{c_0}) = c_n$. Obviously, $p \in P$, otherwise r would not belong to $\gamma_{io}(P)$.

$p = \langle c_0, \mathit{inf}(c)\rangle$ In this case, it exists $r \in \gamma_{io}(P)$ such that $r{\Downarrow}_{c_0} = c_0 \ldots c_i \ldots$ and $\otimes_{i \geq 0} = c$. It is easy to notice that $p \in P$, otherwise, by using γ_{io}, we would not obtain r.

Proposition 3.2.4. *Let $D \in \mathbf{D}_{\mathbf{C}}^{\Pi}$ and $A \in \mathbf{A}_{\mathbf{C}}^{\Pi}$. Then, $\alpha_{io}(\mathcal{P}[\![D\,.\,A]\!]) = \mathcal{B}^{io}[\![D\,.\,A]\!]$.*

Proof.

Consider $D \in \mathbf{D}_{\mathbf{C}}^{\Pi}$ and $A \in \mathbf{A}_{\mathbf{C}}^{\Pi}$, then $\alpha_{io}(\mathcal{P}[\![D.A]\!]) = \mathcal{B}^{io}[\![D.A]\!]$. We show the two inclusions independently.

$\subseteq$ Let $r \in \mathcal{P}[\![D\,.\,A]\!]$ and $c_0 \in \mathbf{C}$ such that $r\Downarrow_{c_0}$ is defined. In order to show that $\alpha_{io}(\{r\}) \subseteq \mathcal{B}^{io}[\![D\,.\,A]\!]$, we distinguish two cases.

1. In case $r\Downarrow_{c_0}$ is finite, by (3.2.1), $\alpha_{io}(\{r\}) = \langle c_0, \mathit{fin}(c_n)\rangle \in \alpha_{io}(\mathcal{P}[\![D\,.\,A]\!])$, where $c_n := \mathit{last}(r\Downarrow_{c_0})$. Moreover, by Definitions 3.1.20, 3.1.16 and 3.1.25, it is easy to notice that r must be of one of the following forms:

   (a) r ends with $\boxtimes$,

   (b) r contains a *stutt* or

   (c) r contains a conditional store $\eta \rightarrowtail d$ such that there is no *stutt* before it and $c_0 \otimes d = \mathit{false}$.

   Now, let us show that on the behavioral part, when A, with initial store c_0, behaves as $\langle A, c_0\rangle \to^* \langle A_n, c_n\rangle \not\to$ (the sequence is finite), r takes also one of those forms. Looking at the agent semantics $\mathcal{A}$ (Definition 3.1.16) we observe that:

   (a) we obtain a sequence that ends with $\boxtimes$ if a subagent of A is equal to skip or tell, this means that, starting from an initial store c_0 such that $\mathit{last}(r\Downarrow_{c_0})$ is well defined, the operational semantics cannot perform any step from the reached configuration $\langle \mathsf{skip}, c_n\rangle \not\to$;

   (b) when A contains an agent $\sum_{i=1}^{n} \mathsf{ask}(g_i) \to A_i$ and $\forall i \in [1,n].\, g_i \neq \mathit{false}$, then a $\mathit{stutt}(\cup_{i=1}^{n})$ is introduced. Since we assume that $r\Downarrow_{c_0}$ is well defined, it holds that the guards are not entailed by c_0 (merged with the store produced by the sequence up to that position), thus the operational semantics cannot perform any step from the reached configuration $\langle \sum_{i=1}^{n} \mathsf{ask}(g_i) \to A_i, c_n\rangle \not\to$;

   (c) when r contains a conditional state $\eta \rightarrowtail d$ (that occurs before any *stutt*) such that $c_0 \otimes d = \mathit{false}$, we can deduce that, starting from $\langle A, c_0\rangle$, we reach in a finite number of operational steps the state $\langle A_n, \mathit{false}\rangle \not\to$, from which no further derivation is possible since an inconsistent store has been produced.

   Thus, by Definition 3.2.2, $\langle c_0, \mathit{fin}(c_n)\rangle \in \mathcal{B}^{io}[\![D\,.\,A]\!]$.

2. In case $r\Downarrow_{c_0} = c_0 \cdots c_i \cdots$ is infinite, let us define $c := \otimes_{i\geq 0} c_i$. By (3.2.1), $\alpha_{io}(\{r\}) = \langle c_0, \mathit{inf}(c)\rangle \in \alpha_{io}(\mathcal{P}[\![D\,.\,A]\!])$. By Theorem 3.1.26, it is easy to notice that $r\Downarrow_{c_0} \in \mathcal{B}^{ss}[\![D\,.\,A]\!]_{c_0}$, in fact, agent A with initial store c_0 behaves in the following way: $\langle A, c_0\rangle \to \ldots \to \langle A_i, c_i\rangle \to \ldots$. By Definition 3.2.2, it follows that $\langle c_0, \mathit{inf}(c)\rangle \in \mathcal{B}^{io}[\![D\,.\,A]\!]$.

$\supseteq$ Let $p \in \mathbf{IO}$, we show that $p \in \mathcal{B}^{io}[\![D\,.\,A]\!] \Rightarrow p \in \alpha_{io}(\mathcal{P}[\![D\,.\,A]\!])$. Let us distinguish two cases.

<u>$p = \langle c_0, \mathit{fin}(c_n)\rangle$</u> By Definition 3.2.2, it follows that $\langle A, c_0\rangle \to \ldots \to \langle A_n, c_0\rangle \not\to$, and by Definition 3.1.1, $c_0 \cdots c_n \in \mathcal{B}^{ss}[\![D\,.\,A]\!]_{c_0}$. By Theorem 3.1.26, it exists $r \in \mathcal{P}[\![D . A]\!]$ such that $r{\Downarrow_{c_0}} = c_0 \cdots c_n$, and by (3.2.1) it follows that $\langle c_0, \mathit{fin}(c_n)\rangle \in \alpha_{io}(\mathcal{P}[\![D\,.\,A]\!])$.

<u>$p = \langle c_0, \mathit{inf}(c)\rangle$</u> By Definition 3.2.2, it follows that $\langle A, c_0\rangle \to \ldots \to \langle A_i, c_i\rangle \to$, and by Definition 3.1.1, $c_0 \cdots c_i \cdots \in \mathcal{B}^{ss}[\![D\,.\,A]\!]_{c_0}$. By Theorem 3.1.26, it exists $r \in \mathcal{P}[\![D . A]\!]$ such that $r{\Downarrow_{c_0}} = c_0 \cdots c_i \cdots$, and by (3.2.1) it follows that $\langle c_0, \mathit{inf}(c)\rangle \in \alpha_{io}(\mathcal{P}[\![D\,.\,A]\!])$.

Theorem 3.2.6. *Let P_1 and P_2 be two tccp programs such that no trace in $\mathcal{P}[\![P_1]\!] \sqcup \mathcal{P}[\![P_2]\!]$ is a failed conditional trace. Then, $\mathcal{O}^{io}[\![P_1]\!] = \mathcal{O}^{io}[\![P_2]\!]$ if and only if $\mathcal{B}^{io}_F[\![P_1]\!] = \mathcal{B}^{io}_F[\![P_2]\!]$.*

Proof.

From Proposition 3.2.4 and by definition of π_F (Definition 3.2.2), for each *tccp* program P, $\pi_F(\alpha_{io}(\mathcal{P}[\![P]\!])) = \mathcal{B}^{io}_F[\![P]\!]$. Thus, it is sufficient to show that $\mathcal{O}^{io}[\![P_1]\!] = \mathcal{O}^{io}[\![P_2]\!] \iff \pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!])) = \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!]))$ for P_1 and P_2 *tccp* programs such that no trace in $\mathcal{P}[\![P_1]\!] \sqcup \mathcal{P}[\![P_2]\!]$ is a failed conditional trace. We prove the two directions separately.

$\Rightarrow$ We prove the equivalent implication:

$$\pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!])) \neq \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!])) \Rightarrow \mathcal{O}^{io}[\![P_1]\!] \neq \mathcal{O}^{io}[\![P_2]\!].$$

Let us assume, without loss of generality, that $\pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!])) \subset \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!]))$, which means that there exist $r_2 \in \mathcal{P}[\![P_2]\!]$ and $c_0 \in \mathbf{C}$ such that $r_2{\Downarrow_{c_0}} = c_0 \cdots c_n$, but it does not exist $r_1 \in \mathcal{P}[\![P_1]\!]$ such that $r_1{\Downarrow_{c_0}} = c_0 \cdots c_n$. Furthermore, $c_n \neq \mathit{false}$ since, by hypothesis, r_2 is not a failed conditional trace. By Theorem 3.1.26, $c_0 \cdots c_n \in \mathcal{B}^{ss}[\![P_2]\!]$ and, by Definition 3.2.2, $\langle c_0, c_n\rangle \in \mathcal{B}^{io}_F[\![P_2]\!]$. Since $\mathcal{B}^{io}_F$ and $\mathcal{O}^{io}$ differ only on sequences terminating in *false* and $c_n \neq \mathit{false}$, it follows that $\langle c_0, c_n\rangle \in \mathcal{O}^{io}[\![P_2]\!]$. On the other hand, we have that $c_0 \cdots c_n \notin \mathcal{B}^{ss}[\![P_1]\!]$, thus $\langle c_0, c_n\rangle \notin \mathcal{B}^{io}_F[\![P_1]\!]$. It is easy to see that, given a *tccp* program P, $\mathcal{O}^{io}[\![P]\!] \subseteq \mathcal{B}^{io}_F[\![P]\!]$, thus it holds that $\langle c_0, c_n\rangle \notin \mathcal{O}^{io}[\![P_1]\!]$. This means that $\langle c_0, c_n\rangle \in \mathcal{O}^{io}[\![P_2]\!] \smallsetminus \mathcal{O}^{io}[\![P_1]\!]$ and we can conclude that $\mathcal{O}^{io}[\![P_1]\!] \neq \mathcal{O}^{io}[\![P_2]\!]$.

$\Leftarrow$ We prove the equivalent implication:

$$\mathcal{O}^{io}[\![P_1]\!] \neq \mathcal{O}^{io}[\![P_2]\!] \Rightarrow \pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!])) \neq \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!])).$$

Without loss of generality, assume that $\mathcal{O}^{io}[\![P_1]\!] \subset \mathcal{O}^{io}[\![P_2]\!]$, thus, there exists $\langle c_0, c_n\rangle \in \mathcal{O}^{io}[\![P_2]\!]$ such that $\langle c_0, c_n\rangle \notin \mathcal{O}^{io}[\![P_1]\!]$. Since no trace in $\mathcal{P}[\![P_1]\!] \sqcup \mathcal{P}[\![P_2]\!]$ is failed, we can assume that $c_n \neq \mathit{false}$. This means that, by using the transition relation defined in [43], we have a derivation of the form $\langle A_2, c_0\rangle \to \ldots \langle A_2', c_n\rangle \not\to$, with $A_2, A_2' \in \mathbf{A}^{\Pi}_{\mathbf{C}}$, $D_2 \in \mathbf{D}^{\Pi}_{\mathbf{C}}$ and $P_2 = D_2\,.\,A_2$; On the other hand, it can be noticed that, by using the transition relation of Figure 2.2, for P_1 there is no derivation starting with c_0 and ending in c_n. Thus, we have that $\langle c_0, c_n\rangle \in \mathcal{B}^{io}_F[\![P_2]\!]$ and $\langle c_0, c_n\rangle \notin \mathcal{B}^{io}_F[\![P_1]\!]$. From Proposition 3.2.4, it follows that $\langle c_0, c_n\rangle \in \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!])) \smallsetminus \pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!]))$ and we can conclude that $\pi_F(\alpha_{io}(\mathcal{P}[\![P_1]\!])) \neq \pi_F(\alpha_{io}(\mathcal{P}[\![P_2]\!]))$.

---

4 Abstract Diagnosis for tccp based on constraint system abstractions

Abstract

We present a generic abstract diagnosis framework for *tccp* programs. This is an effective and completely automatic debugging methodology based on abstract interpretation and parametric to an abstract semantics modeling the properties of interest.

We associate to programs an abstract semantics defined as the least fixpoint of a (monotonic) immediate consequence operator. Then, given the approximated intended behavior of the program, we derive a finitely terminating bottom-up diagnosis method which can be used statically to find discrepancies between the abstract behavior of the program and the intended one. It is shown that these "abstract" discrepancies reflect possible errors in the "concrete" behavior of the program.

We also present an instance of this method based on the abstraction of the underlying constraint system. The elements of the abstract domain are abstract compact sequences which contain approximated information in the conditions and in the stores and collapse in an unique state all the consecutive states that become equal after the abstraction.

Finding program bugs is a long-standing problem in software construction. In the concurrent paradigm, the problem is even worse and the traditional tracing techniques become almost useless. In fact, in presence of concurrency, an error can arise from the interaction of some agents running in parallel. This, on one hand, makes computations not replicable in practice and, on the other hand, complicates the task of locating the exact position of the bug. In the context of *tccp* the presence of timing features adds further difficulties in the debugging phase, since bugs can arise from a synchronization error between the agents running in parallel. Moreover, the presence of non-determinism further complicates the task of finding bugs in *tccp* programs.

There has been a lot of work on algorithmic debugging [113] for declarative languages, which could be a valid proposal for concurrent paradigms, but little effort has been done for the particular case of the concurrent constraint paradigm (*ccp* in short; [104]).

In this chapter, we develop an abstract diagnosis method for *tccp* using the ideas of [25]. Abstract diagnosis was first developed for logic programming in [25] and later it has been applied to other paradigms [1, 8, 51]. This research revealed that a key point for the efficacy of the resulting debugging methodology is the adequacy of the concrete semantics. Thus, in this thesis, much effort has been dedicated to the development of an appropriate concrete semantics for the *tccp* language to start with (see Section 3.1 in Chapter 3).

In order to achieve an effective method, abstract interpretation is used to approximate the concrete semantics and, because of the abstraction, results may be less precise than those that would be obtained by using the concrete semantics itself. By using suitable abstract domains, specific details of the computation can be hidden and, thus, the information that is required to the user about the (abstract) intended behavior can be dramatically reduced. Obviously, if we use more abstract domains we can detect less errors. Furthermore, by using a more expressive abstract domain, we gain in precision, but the specification phase become more complicated and error-prone. Thus, the choice of an abstract domain is often a tradeoff between the precision of errors that can be detected and the effort in providing the specification.

Abstract diagnosis is parametric w.r.t. an abstract program property of interest modeled in a suitable abstract domain **A** and it is *inherently* based on the use of a correct approximation $\mathcal{D}^\alpha$ of the concrete immediate consequence operator $\mathcal{D}$ (see Definition 3.1.20). $\mathcal{D}^\alpha$ evaluates a program in the abstract domain **A** by focusing only in the information about the considered abstract property and by abstracting away from the other program details.

We show that, given the abstract intended specification (written in **A**), we can check the correctness of a *tccp* set of declarations D by a single application of $\mathcal{D}^\alpha[\![D]\!]$ and thus, by a simple static test, we can determine all the declarations which are wrong w.r.t. the considered abstract property.

The diagnosis is based on the detection of *incorrect rules* and *uncovered elements*, both defined in terms of one application of $\mathcal{D}^\alpha[\![D]\!]$ to the abstract specification. It is worth noting that no fixpoint computation is required, since the abstract semantics does not need to be computed.

Thanks to the expressiveness of our concrete semantics, our method is defined on the full *tccp* language. Therefore, it is able to deal with the constructors that introduce non-monotonic behaviors: the ask, now and hiding agents.

This chapter is organized as follows. In Section 4.1, a general abstract diagnosis methodology for *tccp* is presented by using the semantics defined in Section 3.1. In Section 4.2 an abstract domain able to model finite and infinite *tccp* computations is introduced and formally related with the domain of (concrete) conditional traces by means of a Galois Insertion. The elements of the abstract domain are abstract compact sequences which contain approximated information in the conditions and in the stores and collapse in an unique state all the consecutive states that become equal after the abstraction. In Section 4.3, an abstract semantics is induced from the concrete semantics (Section 3.1) and from the defined Galois Insertion. In Section 4.4 some examples of abstract diagnosis of *tccp* programs are illustrated by using the induced abstract semantics of Section 4.3. All the technical proofs of the results can be found in the chapter appendix 4.A.

4.1 Abstract Diagnosis for *tccp* based on Galois Insertions

In this section, following the approach of [25], we define a general abstract diagnosis methodology for *tccp* starting from the concrete semantics $\mathcal{D}$ defined in Section 3.1. This approach is parametric to a Galois Insertion between the domain **M** of conditional traces and an abstract domain **A** chosen to model the property of interest: $(\mathbf{M}, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbf{A}, \leq)$. Let us recall from Definition 1.3.1 that the abstract domain has to be a complete lattice on the form $(\mathbf{A}, \leq, \bigvee, \bigwedge, \top, \bot)$. This abstraction can be systematically lifted to a Galois Insertion $\mathbf{I} \underset{\bar{\alpha}}{\overset{\bar{\gamma}}{\leftrightarrows}} [\mathbf{PC} \to \mathbf{A}]$ by function composition (i.e., $\bar{\alpha}(f) = \alpha \circ f$). In the following we denote as $\mathbf{I_A}$ the domain of abstract interpretations $[\mathbf{PC} \to \mathbf{A}]$.

As explained in Section 1.3, the optimal abstract version of $\mathcal{D}^\alpha$ is defined simply as $\mathcal{D}^\alpha[\![D]\!] := \bar{\alpha} \circ \mathcal{D}[\![D]\!] \circ \bar{\gamma}$ guaranteeing that $\mathcal{D}^\alpha$ is a correct approximation of $\mathcal{D}$ and $\mathcal{F}^\alpha := \mathcal{D}^\alpha[\![D]\!]{\uparrow}\omega$ is the best correct approximation of $\mathcal{F}$ by construction. We recall that correct means $\alpha(\mathcal{F}[\![D]\!]) \leq \mathcal{F}^\alpha[\![D]\!]$ and best means that it is the minimum (w.r.t. $\leq$) of all correct approximations.

Now, following the ideas of [25], we define the abstract diagnosis of *tccp*. The framework of abstract diagnosis comes from the idea of considering the abstract versions of Park's Induction Principle[1]. It can be considered as an extension of declarative debugging since there are instances of the framework that deliver the same results. In general, diagnosing w.r.t. *abstract* properties relieves the user from having to specify in excessive detail the program behavior (which could be more error-prone than the coding itself).

Let us now introduce the workset of abstract diagnosis by starting from the definition of correct and complete set of declarations.

Definition 4.1.1 *Given a set of declarations D and $\mathcal{S}^\alpha \in \mathbf{A}$, which is the specification of the intended behavior of D w.r.t. the property α, we say that*

1. *D is (abstractly)* partially correct *w.r.t. $\mathcal{S}^\alpha$ if $\alpha(\mathcal{F}[\![D]\!]) \leq \mathcal{S}^\alpha$.*
2. *D is (abstractly)* complete *w.r.t. $\mathcal{S}^\alpha$ if $\mathcal{S}^\alpha \leq \alpha(\mathcal{F}[\![D]\!])$.*
3. *D is* totally correct *w.r.t. $\mathcal{S}^\alpha$, if it is partially correct and complete.*

It is worth noting that the above definition is given in terms of the abstraction of the concrete semantics $\alpha(\mathcal{F}[\![D]\!])$ and not in terms of the (possibly less precise) abstract semantics $\mathcal{F}^\alpha[\![D]\!]$. Note that $\mathcal{S}^\alpha$ is the abstraction of the intended concrete semantics of D. Thus, the user can only reason in terms of the properties of the expected concrete semantics without being concerned with (approximate) abstract computations. The *diagnosis* determines the "originating" symptoms and, in the case of incorrectness, the relevant process declaration in the program. This is captured by the definitions of *abstractly incorrect process declaration* and *abstract uncovered element*:

Definition 4.1.2 *Let $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$, R a process declaration for process p, $e \in \mathbf{A}$ and $\mathcal{S}^\alpha \in \mathbf{I_A}$.*

- *R is* abstractly incorrect *w.r.t. $\mathcal{S}^\alpha$ (on testimony e) if $e \leq \mathcal{D}^\alpha[\![\{R\}]\!]_{\mathcal{S}^\alpha}(p(\vec{x}))$ and $e \wedge \mathcal{S}^\alpha(p(\vec{x})) = \bot$.*
- *e is an* uncovered element *for $p(\vec{x})$ w.r.t. $\mathcal{S}^\alpha$ if $e \leq \mathcal{S}^\alpha(p(\vec{x}))$ and $e \wedge \mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha}(p(\vec{x})) = \bot$.*

Informally, R is abstractly incorrect if it derives a wrong abstract element e from the intended semantics. Dually, e is uncovered if the declarations cannot derive it from the intended semantics.

It is worth noting that the notions of correctness and completeness are defined in terms of $\alpha(\mathcal{F}[\![D]\!])$, i.e., in terms of abstraction of the concrete semantics. The abstract

[1] A concept of formal verification that is undecidable in general.

version of algorithmic debugging [113], which is based on symptoms (i.e., deviations between $\alpha(\mathcal{F}[\![D]\!])$ and $\mathcal{S}^\alpha$), requires the construction of $\alpha(\mathcal{F}[\![D]\!])$ and therefore a fixpoint computation. In contrast, the notions of abstractly incorrect process declarations and abstract uncovered elements are defined in terms of *just one* application of $\mathcal{D}^\alpha[\![D]\!]$ to $\mathcal{S}^\alpha$. The issue of the precision of the abstract semantics is specially relevant in establishing the relation between the two concepts, i.e., the relation between abstractly incorrect process declarations and abstract uncovered elements on one side, and abstract partial correctness and completeness, on the other side.

Theorem 4.1.3 *Let $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$ and $\mathcal{S}^\alpha \in \mathbf{I_A}$.*

1. *If there are no abstractly incorrect process declarations in D (i.e., $\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$), then D is partially correct w.r.t. $\mathcal{S}^\alpha$ (i.e., $\alpha(\mathcal{F}[\![D]\!]) \leq \mathcal{S}^\alpha$).*

2. *Let D be partially correct w.r.t. $\mathcal{S}^\alpha$. If D has abstract uncovered elements then D is not complete (i.e., $\mathcal{S}^\alpha \nleq \alpha(\mathcal{F}[\![D]\!])$).*

Absence of abstractly incorrect declarations is a sufficient condition for partial correctness, but it is not necessary. When applying the diagnosis w.r.t. approximate properties, the results may be weaker than those that can be achieved on concrete domains just because of approximation. For this reason, it can happen that a (concretely) correct declaration is abstractly incorrect. Hence, abstract incorrect declarations are in general just a warning about a possible source of errors.

However, an abstract correct declaration cannot contain an error; thus, no (manual) inspection is needed for declarations which are not abstractly incorrect. Moreover, as shown by the following theorem, all concrete errors—that are "visible"—are detected, as they lead to an abstract incorrectness or abstract uncovered.

Theorem 4.1.4 *Let r be a process declaration and $\mathcal{S}$ a concrete specification.*

1. *If $\mathcal{D}[\![\{r\}]\!]_{\mathcal{S}} \nleq \mathcal{S}$ and $\alpha(\mathcal{D}[\![\{r\}]\!]_{\mathcal{S}}) \nleq \alpha(\mathcal{S})$ then r is abstractly incorrect w.r.t. $\alpha(\mathcal{S})$.*

2. *If there exists an abstract uncovered element a w.r.t. $\alpha(\mathcal{S})$, such that $\gamma(a) \sqsubseteq \mathcal{S}$ and $\gamma(\bot) = \{\epsilon\}$, then there exists a concrete uncovered element c w.r.t. $\mathcal{S}$ (i.e., $c \sqsubseteq \mathcal{S}$ and $c \sqcap \mathcal{D}[\![D]\!]_{\mathcal{S}} = \{\epsilon\}$).*

The principal results of abstract diagnosis can be summarized by the following points:

- absence of abstractly incorrect rules implies partial correctness,
- every incorrectness error is identified by an abstractly incorrect rule,
- an abstract incorrect rule does not always correspond to a bug (it is just a warning),
- there does not exist a sufficient condition for completeness.

It is important to note that this method is correct by construction because it has been derived by applying abstract interpretation techniques properly.

In the following, we present an abstraction scheme for the domain $\mathbf{M}$ and the corresponding induced abstract semantics for *tccp* which is obtained by using standard abstract interpretation techniques. This semantics will be used to instantiate the abstract diagnosis framework and achieve a fully-automatic debugging methodology for *tccp*.

4.2 Abstraction scheme

In this section we present an abstraction scheme for *tccp* computations, i.e., maximal sets of conditional traces. This abstraction is defined by successive lifting. We start with a function that abstracts the information component of the program semantics, i.e., the constraints, then, we build the abstraction of conditional states, then of conditional traces and, finally, of maximal sets.

The result of the abstraction is a maximal set of abstract conditional traces. An abstract conditional trace contains approximated information in its conditions and stores, furthermore, consecutive conditional states that become identical after the approximation are collapsed together in a unique state.

We define a Galois Insertion that formalizes this approximation by relating the domain of concrete conditional traces with the domain of abstract ones.

4.2.1 Constraint System Abstraction

In the semantics of Section 3.1, constraints can assume different meanings depending on the role they play within a conditional state. On one hand, positive conditions and stores represent the constraints that are entailed by the store. For this reason, these constraints can be considered as the positive information in the trace. On the other hand, constraints contained in negative conditions and stuttering constructs are those that must not be entailed in order to make the computation proceed, thus, they can be seen as the negative information in the trace.

As a consequence, in order to approximate correctly the information in the conditional traces, it is necessary to define two different approximating functions for the (concrete) underlying constraint system $\mathbf{C} = \langle \mathcal{C}, \leq, \otimes, \oplus, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$:

- an over-approximating function τ^+ for the positive information, and
- an under-approximating function τ^- for the negative information.

The over-approximating function $\tau^+ \colon \mathbf{C} \to \hat{\mathbf{C}}$ maps a concrete constraint into an upper-abstract constraint $\hat{\mathbf{C}} = \langle \hat{\mathcal{C}}, \hat{\leq}, \hat{\otimes}, \hat{\oplus}, \widehat{\mathit{false}}, \widehat{\mathit{true}}, \mathit{Var}, \hat{\exists} \rangle$. Similarly, the under-approximating function $\tau^- \colon \wp(\mathbf{C}) \to \check{\mathbf{C}}$ maps a set of concrete constraints into a lower-abstract constraint system $\check{\mathbf{C}} = \langle \check{\mathcal{C}}, \check{\leq}, \check{\otimes}, \check{\oplus}, \check{\mathit{false}}, \check{\mathit{true}}, \mathit{Var}, \check{\exists} \rangle$. We often use the inverse relations $\hat{\vdash}$ and $\check{\vdash}$ instead of $\hat{\leq}$ and $\check{\leq}$, respectively.

We define two "external" operations $\hat{\times} \colon \mathbf{C} \times \hat{\mathbf{C}} \to \hat{\mathbf{C}}$ and $\check{\times} \colon \mathbf{C} \times \check{\mathbf{C}} \to \check{\mathbf{C}}$ that update an abstract constraint with the information contained in a concrete constraint. In addition, a "bridge" relation $\tilde{\vdash} \in \hat{\mathbf{C}} \times \check{\mathbf{C}}$ is introduced in order to decide if an upper-abstract constraint is consistent with a lower-abstract constraint.

Abstract and concrete constraint systems are related by the following conditions. Given $c, c', a, b \in \mathbf{C}$ and $C, C' \subseteq \mathbf{C}$,

$$c \mathbin{\hat{\times}} \tau^+(a) = \tau^+(c \otimes a) \tag{4.2.1a}$$

$$c \mathbin{\check{\times}} \tau^-(C) = \tau^-(\{c\} \cup C) \tag{4.2.1b}$$

$$\tau^+(a \otimes b) = \tau^+(a) \mathbin{\hat{\otimes}} \tau^+(b) \tag{4.2.1c}$$

$$\tau^-(C \cup C') = \tau^-(C) \mathbin{\check{\oplus}} \tau^-(C') \tag{4.2.1d}$$

$$c \vdash c' \implies \tau^+(c) \mathrel{\hat{\vdash}} \tau^+(c') \tag{4.2.1e}$$

$$C \subseteq C' \Longrightarrow \tau^{-}(C) \mathrel{\check{\vdash}} \tau^{-}(C') \tag{4.2.1f}$$

$$\tau^{+}(\exists_x a) = \hat{\exists}_x \tau^{+}(a) \tag{4.2.1g}$$

$$\tau^{-}(\{\exists_x c \mid c \in C\}) = \check{\exists}_x \tau^{-}(C) \tag{4.2.1h}$$

$$\forall c \in C.\ a \nvdash c \iff \tau^{+}(a) \mathrel{\tilde{\nvdash}} \tau^{-}(C) \tag{4.2.1i}$$

The first two conditions establish the relation between the “external” operations and the merge and join of the concrete constraint systems. Then, conditions (4.2.1c) and (4.2.1d) state that the $\hat{\otimes}$ (respectively $\check{\otimes}$) operator must be precise w.r.t. the τ^{+} (respectively τ^{-}) abstractions. Condition (4.2.1e) says that the over-approximation is correct, in the sense that, if two concrete stores are related, then such relation is preserved in the over-approximation (τ^{+}). Properties (4.2.1g) and (4.2.1h) relate the concrete hiding operator with the abstract ones in the expected way. Finally, the last condition is very important since it makes explicit the relation between the two abstractions. It says that a given concrete store does not satisfy any of those in a given set C if and only if its over-approximation cannot satisfy (by means of the “bridge” relation) the under-approximation of the set.

It follows directly from (4.2.1e) that $\tau^{+}(\mathit{true}) = \hat{\mathit{true}}$ (since $\forall c \in \mathbf{C},\ c \vdash \mathit{true} \Longrightarrow \tau^{+}(c) \mathrel{\hat{\vdash}} \tau^{+}(\mathit{true})$) and $\tau^{+}(\mathit{false}) = \hat{\mathit{false}}$ (since $\forall c \in \mathbf{C},\ \mathit{false} \vdash c \Longrightarrow \tau^{+}(\mathit{false}) \mathrel{\hat{\vdash}} \tau^{+}(c)$).

Furthermore, by (4.2.1f), $\tau^{-}(\varnothing) = \check{\mathit{false}}$ (since $\forall C \subseteq \mathbf{C},\ \varnothing \subseteq C \Longrightarrow \tau^{-}(\varnothing) \mathrel{\check{\vdash}} \tau^{-}(C)$) and $\tau^{-}(\mathcal{C}) = \check{\mathit{true}}$ (since $\forall C \subseteq \mathbf{C},\ C \subseteq \mathcal{C} \Longrightarrow \tau^{-}(C) \mathrel{\check{\vdash}} \tau^{-}(\mathcal{C})$).

Let us show some examples that make explicit the kind of constraint system abstractions we are interested in.

Example 4.2.1 (Finite Domain Abstraction)

Consider the concrete constraint system $\mathbf{L} := \langle \mathcal{L}, \Leftarrow, \wedge, \vee, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$ defined in 1.4.3 and suppose we are interested only in the variables whose value belongs to a given range of natural numbers. We introduce an upper-abstract constraint system $\hat{\mathbf{FD}}(n) := \langle \mathcal{FD}(n), \Leftarrow, \wedge, \vee, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$, where $\mathcal{FD}(n) := \{x \mathrel{\hat{=}} k \mid x \in \mathit{Var},\ 0 \leq k \leq n-1\}$. As illustrated in [87], this constraint system provides a theory of variables ranging over a finite domain of values $\{0, \ldots, n-1\}$.

The Hasse diagram of this abstract constraint system is:

$$\begin{tikzcd}
 & & \mathit{false} \arrow[dll, dash] \arrow[dl, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
x \mathrel{\hat{=}} 0 & x \mathrel{\hat{=}} 1 & \cdots\cdots\cdots & x \mathrel{\hat{=}} n-2 & x \mathrel{\hat{=}} n-1 \\
 & & \mathit{true} \arrow[ull, dash] \arrow[ul, dash] \arrow[ur, dash] \arrow[urr, dash] & &
\end{tikzcd}$$

The abstract over-approximating function τ^{+} which relates $\mathbf{L}$ and $\hat{\mathbf{FD}}(n)$ is defined by cases:

$$\tau^{+}(x = k) = \begin{cases} x \mathrel{\hat{=}} k & \text{if } 0 \leq k < n \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\tau^{+}(x < k) = \begin{cases} (x \mathrel{\hat{=}} 0) \vee \cdots \vee (x \mathrel{\hat{=}} k-1) & \text{if } 0 < k \leq n \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\tau^+(x \le k) = \begin{cases} (x \mathrel{\hat{=}} 0) \vee \cdots \vee (x \mathrel{\hat{=}} k) & \text{if } 0 \le k < n \\ \mathit{false} & \text{otherwise} \end{cases}$$
$$\tau^+(x > k) = \begin{cases} (x \mathrel{\hat{=}} k+1) \vee \cdots \vee (x \mathrel{\hat{=}} n-1) & \text{if } 0 \le k < n-1 \\ \mathit{false} & \text{otherwise} \end{cases}$$
$$\tau^+(x \ge k) = \begin{cases} (x \mathrel{\hat{=}} k) \vee \cdots \vee (x \mathrel{\hat{=}} n-1) & \text{if } 0 \le k < n \\ \mathit{false} & \text{otherwise} \end{cases}$$
$$\tau^+(\mathit{false}) = \mathit{false}$$
$$\tau^+(\mathit{true}) = \mathit{true}$$

The corresponding lower-abstract constraint system is $\check{\mathbf{FD}}(n) := \langle \wp(\mathcal{FD}(n)), \subseteq, \cup, \cap, \mathcal{FD}(n), \varnothing, \mathit{Var}, \exists \rangle$ with the associated function τ^- defined as $\tau^-(C) = \bigcup_{c \in C} \tau^+(c)$.

The external operator $\hat{\times}$ is defined as $c \mathbin{\hat{\times}} \hat{d} = \tau^+(c) \wedge \hat{d}$, while $\check{\times}$ is defined as $c \mathbin{\check{\times}} \check{d} = \tau^-(\{c\}) \cup \check{d}$. Therefore, it follows directly that the conditions (4.2.1a) and (4.2.1b) hold. The bridge relation $\tilde{\vdash}$ is defined as $\hat{a} \mathrel{\tilde{\vdash}} \check{B} \iff \exists \hat{b} \in \check{B}.\ \hat{a} \Rightarrow \hat{b}$, however, we usually deal with the negative relation $\tilde{\nvdash}$: $\hat{a} \mathrel{\tilde{\nvdash}} \check{B} \iff \forall \hat{b} \in \check{B}.\ \hat{a} \not\Rightarrow \hat{b}$.

It is easy to check that also the rest of conditions hold.

Let us show some examples for the conditions regarding precision (4.2.1c) and (4.2.1d). We choose $n = 5$, thus the abstract domain is $\hat{\mathbf{FD}}(5)$. Let $x = 2$ and $x = 4$ be a and b in condition (4.2.1c):

$$\tau^+(x = 2 \wedge x = 4) = \tau^+(\mathit{false}) = \mathit{false}$$
$$\tau^+(x = 2) \wedge \tau^+(x = 4) = x \mathrel{\hat{=}} 2 \wedge x \mathrel{\hat{=}} 4 = \mathit{false}$$

Now, let $x > 2$ and $x > 3$ be a and b in condition (4.2.1c):

$$\tau^+(x > 2 \wedge x > 3) = \tau^+(x > 3) = (x \mathrel{\hat{=}} 4) \vee (x \mathrel{\hat{=}} 5)$$
$$\tau^+(x > 2) \wedge \tau^+(x > 3) = \big((x \mathrel{\hat{=}} 3) \vee (x \mathrel{\hat{=}} 4) \vee (x \mathrel{\hat{=}} 5)\big) \wedge \big((x \mathrel{\hat{=}} 4) \vee (x \mathrel{\hat{=}} 5)\big) = (x \mathrel{\hat{=}} 4) \vee (x \mathrel{\hat{=}} 5)$$

Thus, in these examples the condition (4.2.1c) holds. It also holds when the abstraction of one of the elements is *false*:

$$\tau^+(x = 2 \wedge x = 7) = \tau^+(\mathit{false}) = \mathit{false}$$
$$\tau^+(x = 2) \wedge \tau^+(x = 7) = x \mathrel{\hat{=}} 2 \wedge \mathit{false} = \mathit{false}$$

Now, let $\{x = 2\}$ and $\{x = 3\}$ be C and C' in condition (4.2.1d):

$$\tau^-(\{x = 2,\, x = 3\}) = \{\tau^+(x = 2),\, \tau^+(x = 3)\} = \{x \mathrel{\hat{=}} 2,\, x \mathrel{\hat{=}} 3\}$$
$$\tau^-(\{x = 2\}) \cup \tau^-(\{x = 3\}) = \{\tau^+(x = 2),\, \tau^+(x = 3)\} = \{x \mathrel{\hat{=}} 2,\, x \mathrel{\hat{=}} 3\}$$

Let us consider also the case when in one of the sets C and C' there is an element whose abstraction is *false*:

$$\tau^-(\{x = 2\} \cup \{x = 9\}) = \{\tau^+(x = 2)\} \cup \{\tau^+(x = 9)\} = \{x \mathrel{\hat{=}} 2\} \cup \{\mathit{false}\} = \{x \mathrel{\hat{=}} 2, \mathit{false}\}$$
$$\tau^-(\{x = 2\}) \cup \tau^-(\{x = 9\}) = \{\tau^+(x = 2)\} \cup \{\tau^+(x = 9)\} = \{x \mathrel{\hat{=}} 2\} \cup \{\mathit{false}\} = \{x \mathrel{\hat{=}} 2, \mathit{false}\}$$

Finally, one important condition is that regarding the bridge relation (4.2.1i). Let us show an example for the satisfaction of that condition. Let $\{x = 3, x = 4\}$ and $x = 2$ be, respectively, C and a in condition (4.2.1i). We have that $x = 2 \not\Rightarrow x = 3$ and $x = 2 \not\Rightarrow x = 4$, thus, for all $c \in C$, $a \not\Rightarrow c$. Furthermore, by definition of $\tilde{\nvdash}$, it follows that $x \mathrel{\hat{=}} 2 \mathrel{\tilde{\nvdash}} \{x \mathrel{\hat{=}} 3, x \mathrel{\hat{=}} 4\}$ since $x \mathrel{\hat{=}} 2 \not\Rightarrow x \mathrel{\hat{=}} 3$ and $x \mathrel{\hat{=}} 2 \not\Rightarrow x \mathrel{\hat{=}} 4$. Thus, condition (4.2.1i) is satisfied.

Example 4.2.2 (Positive-Negative Abstraction)

Consider again, the Constraint System 1.4.3 and suppose we are interested only in the sign of the variables. In this case, we design the upper-abstract constraint system in this way: $\mathbf{\hat{PN}} := \langle \mathcal{PN}, \Leftarrow, \wedge, \vee, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$ where $\mathcal{PN} = \{\mathrm{pos}_x, \mathrm{neg}_x \mid x \in \mathit{Var}\} \cup \{\mathit{false}, \mathit{true}\}$.

The Hasse diagram of this abstract constraint system is:

$$\begin{tikzcd}
 & \mathit{false} \arrow[dl, no head] \arrow[dr, no head] & \\
\mathrm{pos}_x \arrow[dr, no head] & & \mathrm{neg}_x \arrow[dl, no head] \\
 & \mathit{true} &
\end{tikzcd}$$

The abstract over-approximation τ^+ is defined by cases as follows:

$$\tau^+(x > a) = \begin{cases} \mathrm{pos}_x & \text{if } a \geq 0 \\ \mathit{false} & \text{otherwise} \end{cases} \qquad \tau^+(x \geq a) = \begin{cases} \mathrm{pos}_x & \text{if } a > 0 \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\tau^+(x < a) = \begin{cases} \mathrm{neg}_x & \text{if } a \leq 0 \\ \mathit{false} & \text{otherwise} \end{cases} \qquad \tau^+(x \leq a) = \begin{cases} \mathrm{neg}_x & \text{if } a < 0 \\ \mathit{false} & \text{otherwise} \end{cases}$$

$$\begin{aligned} \tau^+(l < x < u) &= \mathit{false} \\ \tau^+(x = a) &= \mathit{false} \\ \tau^+(\mathit{false}) &= \mathit{false} \\ \tau^+(\mathit{true}) &= \mathit{true} \end{aligned}$$

This abstraction keeps the information regarding the sign of variables, but not the concrete value of them. The corresponding lower-abstract constraint system is $\mathbf{\check{PN}} := \langle \wp(\mathcal{PN}), \subseteq, \cup, \cap, \mathcal{PN}, \varnothing, \mathit{Var}, \exists \rangle$ with the associated under-approximating function τ^-, defined as $\tau^-(C) = \bigcup_{c \in C} \tau^+(c)$.

The external operator $\hat{\times}$ is defined as $c \mathbin{\hat{\times}} \hat{d} = \tau^+(c) \wedge \hat{d}$, while $\check{\times}$ is defined as $c \mathbin{\check{\times}} \check{d} = \tau^-(\{c\}) \cup \check{d}$. Thus, conditions (4.2.1a) and (4.2.1b) follows immediately. The bridge relation $\tilde{\vdash}$ is defined as $\hat{a} \mathrel{\tilde{\vdash}} \check{B} \iff \exists \hat{b} \in \check{B}.\ \hat{a} \Rightarrow \hat{b}$. It is easy to see that also the rest of conditions hold.

As in the previous example, we illustrate the satisfaction of the most interesting conditions by means of some instantiations. Let $x \geq 2$ and $x > 7$ be a and b in condition (4.2.1c):

$$\begin{aligned} \tau^+(x \geq 2 \wedge x > 7) &= \tau^+(x > 7) = \mathrm{pos}_x \\ \tau^+(x \geq 2) \wedge \tau^+(x > 7) &= \mathrm{pos}_x \wedge \mathrm{pos}_x = \mathrm{pos}_x \end{aligned}$$

Now, consider two cases that reach the *false* abstract constraint:

$$\begin{aligned} \tau^+(x \geq 1 \wedge x = 1) &= \tau^+(x = 1) = \mathit{false} \\ \tau^+(x \geq 1) \wedge \tau^+(x = 1) &= \mathrm{pos}_x \wedge \mathit{false} = \mathit{false} \end{aligned}$$

$$\tau^+(x \geq 4 \wedge x \leq -3) = \tau^+(\mathit{false}) = \mathit{false}$$
$$\tau^+(x \geq 4) \wedge \tau^+(x \leq -3) = \mathrm{pos}_x \wedge \mathrm{neg}_x = \mathit{false}$$

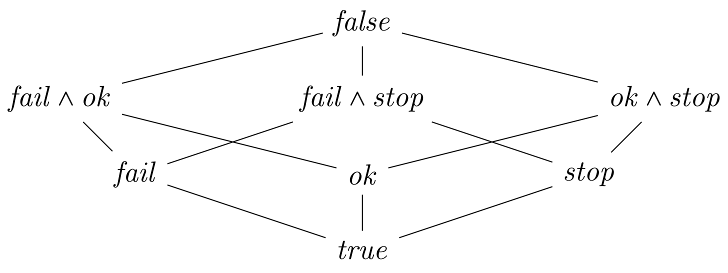
Similarly to the previous example, the condition (4.2.1d) follows immediately from the definition of the function τ^- because we have that $\tau^-(\{c\}) = \{\tau^+(c)\}$ for all $c \in \mathbf{L}$.

Finally, let us show two instances of the condition (4.2.1i). Let $\{x = 2\}$ and $x > 2$ be C and a in the condition, respectively. Then, it is easy to check that $x > 2 \nvdash x = 2$ and $\mathrm{pos}_x \mathrel{\tilde{\nvdash}} \{\mathit{false}\}$. Now, let $C = \{x < -3\}$, then $x > 2 \nvdash x < -3$ and $\mathrm{pos}_x \mathrel{\tilde{\nvdash}} \{\mathrm{neg}_x\}$. Thus, condition (4.2.1i) is satisfied in both cases.

In the following example, we show how to approximate streams, used to model imperative-style variables (see Section 2.3).

Example 4.2.3 (Stream Abstraction)

Consider a domain composed by the elements *fail*, *ok*, *stop*, *true* and *false*. With $\wedge$ we denote the conjunction in this domain. The order relation is the one represented in the following Hasse diagram.



In this case, the concrete constraint system is the Herbrand one (see 1.4.2) where the terms are the values in the given domain. Due to its simplicity, its upper-abstract counterpart coincides with the concrete constraint system itself, while the lower-abstract counterpart is simply the extension of the concrete constraint system to sets of constraints. Therefore, conditions 4.2.1 hold directly.

As already mentioned in Section 2.3, in *tccp* the store is monotonic, thus, we need to use *streams* to model the *imperative-style* variables [43]. In this case we need two levels of abstraction: one for the informational content (the constraints appearing in the stream) and one for the structure of the stream. In order to make this abstraction effective it is necessary to store both the last instantiated value in the stream, and the link between the stream and the name of its tail, which could be instantiated afterward. As in the previous examples, the under-approximation is defined simply as the natural extension of τ^+ to sets of constraints as streams.

For example, the positive abstraction for the stream $s = [\mathit{fail}|s']$ with $s' = [\mathit{ok}|s'']$ is $(s \doteq \mathit{ok}) \wedge (s \bowtie s'')$, where $\doteq$ associates each stream to his last instantiated value, while $\bowtie$ relates each stream with his tail. When also s'' is instantiated, namely $s'' = [\mathit{stop}|s''']$, the abstraction of s becomes $(s \doteq \mathit{stop}) \hat{\otimes} (s \bowtie s''')$.

It can be easily noticed that, if the conditions listed in 4.2.1 hold for the underlying abstract constraint system, they will also hold for the correspondent stream framework.

4.2.2 Abstraction of information in conditional traces

In this section we introduce the abstract version of the conditional traces. In the sequel, all definitions are parametric w.r.t. an upper-abstract cylindric constraint system $\hat{\mathbf{C}} = \langle \hat{\mathcal{C}}, \hat{\preceq}, \hat{\otimes}, \hat{\oplus}, \widehat{false}, \widehat{true}, Var, \hat{\exists} \rangle$ and a lower-abstract cylindric constraint system $\check{\mathbf{C}} = \langle \check{\mathcal{C}}, \check{\preceq}, \check{\otimes}, \check{\oplus}, \check{false}, \check{true}, Var, \check{\exists} \rangle$.

As in Chapter 3, we denote by ϵ the empty sequence and by $s_1 \cdot s_2$ the concatenation of two sequences s_1, s_2. We also abuse notation and, given a set of sequences S, by $s_1 \cdot S$ we denote $\{s_1 \cdot s_2 \,|\, s_2 \in S\}$.

The idea is to associate an (abstract) condition to each state with approximated information in the stores and in the conditions.

Definition 4.2.4 (Abstract condition) *An* abstract condition $\tilde{\eta}$ *over an upper-abstract constraint system* $\hat{\mathbf{C}}$ *and a lower-abstract constraint system* $\check{\mathbf{C}}$ *is a pair* $\tilde{\eta} = (\hat{\eta}, \check{\eta})$ *where*

- $\hat{\eta} \in \hat{\mathbf{C}}$ *is called* abstract positive condition, *and*
- $\check{\eta} \in \check{\mathbf{C}}$ *is called* abstract negative condition.

An abstract condition is valid *when* $\hat{\eta} \neq \widehat{false}$, $\check{\eta} \neq \check{true}$, *and* $\hat{\eta} \mathrel{\tilde{\nvdash}} \check{\eta}$. *We denote* $\tilde{\Lambda}_{\mathbf{C}}$ *the set of all valid conditions and* $\tilde{\Delta}_{\mathbf{C}}$ *the subset of valid ones.*

The conjunction of two abstract conditions $\tilde{\eta}_1 = (\hat{\eta}_1, \check{\eta}_1)$ *and* $\tilde{\eta}_2 = (\hat{\eta}_2, \check{\eta}_2)$ *is defined as* $\eta_1 \mathbin{\hat{\otimes}} \eta_2 := (\eta_1^+ \mathbin{\hat{\otimes}} \eta_2^+, \eta_1^- \mathbin{\check{\oplus}} \eta_2^-)$. *Two abstract conditions are called* incompatible *if their conjunction is not valid.*

An abstract store $\hat{c} \in \hat{\mathbf{C}}$ *is* consistent *with* $\tilde{\eta}$, *written* $\hat{c} \mathrel{\tilde{\gg}} \tilde{\eta}$, *if* $\hat{c} \mathrel{\tilde{\nvdash}} \check{\eta}$ *and* $\hat{c} \mathbin{\hat{\otimes}} \hat{\eta} \neq \widehat{false}$. *Moreover, we say that* $\hat{c}$ satisfies $\tilde{\eta}$, *written* $\hat{c} \mathrel{\tilde{\Vdash}} \tilde{\eta}$, *when* $\hat{c} \mathrel{\hat{\vdash}} \hat{\eta}$ *and* $\hat{c} \mathrel{\tilde{\nvdash}} \check{\eta}$. *We define the existential quantification on conditions as* $\tilde{\exists}_x \tilde{\eta} := (\hat{\exists}_x \hat{\eta}, \check{\exists}_x \check{\eta})$.

Given $c \in \mathbf{C}$ and $(\eta^+, \eta^-) \in \tilde{\Lambda}_{\mathbf{C}}$:

$$(\eta^+, \eta^-) \text{ is valid} \iff (\tau^+(\eta^+), \tau^-(\eta^-)) \text{ is (abstractly) valid} \tag{4.2.2}$$

$$c \gg (\eta^+, \eta^-) \implies \tau^+(c) \mathrel{\tilde{\gg}} (\tau^+(\eta^+), \tau^-(\eta^-)) \tag{4.2.3}$$

$$c \Vdash (\eta^+, \eta^-) \implies \tau^+(c) \mathrel{\tilde{\Vdash}} (\tau^+(\eta^+), \tau^-(\eta^-)) \tag{4.2.4}$$

Those properties follows directly from the definitions of $\gg$ and $\Vdash$ (3.1.2), and from the conditions listed in 4.2.1.

Let us define the abstract version of conditional state and conditional trace.

Definition 4.2.5 (Abstract conditional state) *An* abstract conditional state *over an upper-abstract constraint system* $\hat{\mathbf{C}}$ *and a lower-abstract constraint system* $\check{\mathbf{C}}$, *is one of the following constructs.*

Abstract conditional store. *A pair* $\tilde{\eta} \rightarrowtail \hat{c}$, *for each* $\tilde{\eta} \in \tilde{\Lambda}_{\mathbf{C}}$ *and* $\hat{c} \in \hat{\mathbf{C}}$.

Abstract stuttering. *The construct* $stutt(\check{\eta})$, *for each* $\check{\eta} \in \check{\mathbf{C}}$ *such that* $\check{\eta} \neq \check{true}$.

End of a process. *The construct* $\boxtimes$.

In an abstract conditional store $\tilde{t} = \tilde{\eta} \rightarrowtail \hat{c}$, $\hat{c}$ *is the* abstract store *of* $\tilde{t}$.

We say that $\tilde{\eta} \rightarrowtail \hat{c}$ *is* valid *if* $\tilde{\eta}$ *is valid.*

Definition 4.2.6 (Abstract conditional trace) *An abstract conditional trace is a sequence of abstract conditional states of the form:* $\tilde{t}_1 \dots \tilde{t}_n \dots$*, possibly ended with* $\boxtimes$*, which respects the following properties:*

Monotonicity. *For each* $\tilde{t}_i = \tilde{\eta}_i \rightarrowtail \hat{c}_i$ *and* $\tilde{t}_j = \tilde{\eta}_j \rightarrowtail \hat{c}_j$ *such that* $j \geq i$, $\hat{c}_j \mathrel{\hat{\vdash}} \hat{c}_i$.

Consistency. *For each* $\tilde{t}_i = \tilde{\eta}_i \rightarrowtail \hat{c}_i$ *and* $\tilde{t}_{i+1}$ *either of the form* $(\hat{\eta}_{i+1}, \check{\eta}_{i+1}) \rightarrowtail \hat{c}_{i+1}$ *or* $stutt(\check{\eta}_{i+1})$*, we have that* $\hat{c}_i \mathrel{\tilde{\nvdash}} \check{\eta}_{i+1}$.

We denote by $\mathbf{M}^{\pm}$ the set of all maximal abstract conditional traces.
$(\mathbf{M}^{\pm}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}^{\pm}, \{\epsilon\})$ is the lattice composed by sets of maximal abstract traces. It can be noticed that the order relation is the same defined for the concrete conditional traces.

Let us formalize the relation between concrete conditional traces and abstract conditional traces. Intuitively, we abstract the positive information in the concrete traces with the over-approximation function τ^+ and the negative one with the under-approximation one τ^-. In this way, abstract conditional traces contain only approximated information in abstract conditions and stores.

Definition 4.2.7 (Abstraction of maximal conditional traces) *Let* $\tau^+ \colon \mathbf{C} \to \hat{\mathbf{C}}$ *and* $\tau^- \colon \wp(\mathbf{C}) \to \hat{\mathbf{C}}$ *be, respectively, the over and the under approximating functions. Given* $r \in \mathbf{M}$*, we define its abstraction* $\alpha^{\pm}$ *as follows:*

$$\alpha^{\pm}(\epsilon) = \epsilon \tag{4.2.5}$$
$$\alpha^{\pm}(\boxtimes) = \boxtimes \tag{4.2.6}$$
$$\alpha^{\pm}((\eta^+, \eta^-) \rightarrowtail c \cdot r) = (\tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(c) \cdot \alpha^{\pm}(r) \tag{4.2.7}$$
$$\alpha^{\pm}(stutt(\eta^-) \cdot r) = stutt(\tau^-(\eta^-)) \cdot \alpha^{\pm}(r) \tag{4.2.8}$$

The corresponding concretization function, given an abstract conditional trace returns the associated set of concrete conditional traces:

$$\gamma^{\pm}(\epsilon) = \{\epsilon\} \tag{4.2.9}$$
$$\gamma^{\pm}(\boxtimes) = \{\boxtimes\} \tag{4.2.10}$$
$$\gamma^{\pm}((\hat{\eta}, \check{\eta}) \rightarrowtail \hat{c} \cdot \bar{r}) = \bigsqcup\{(\eta^+, \eta^-) \rightarrowtail c \cdot r \mid \tau^+(\eta^+) = \hat{\eta},\ \tau^-(\eta^-) = \check{\eta},\ \tau^+(c) = \hat{c},\ r \in \gamma^{\pm}(\bar{r})\} \tag{4.2.11}$$
$$\gamma^{\pm}(stutt(\check{\eta}) \cdot \bar{r}) = \bigsqcup\{stutt(\eta^-) \cdot r \mid \tau^-(\eta^-) = \check{\eta},\ r \in \gamma^{\pm}(\bar{r})\} \tag{4.2.12}$$

Let $r \in \mathbf{M}$, it follows directly from property 4.2.4:

$$\text{r is self-sufficient} \implies \tilde{\alpha}(r) \text{ is astractly self-sufficient} \tag{4.2.13}$$

We abuse in notation by calling $(\alpha^{\pm}, \gamma^{\pm})$ the two functions that relate sets of concrete traces to sets of abstract traces in the following way:

$$\alpha^{\pm}(R) = \bigsqcup\{\alpha^{\pm}(r) \mid r \in R\}$$
$$\gamma^{\pm}(\bar{R}) = \bigsqcup\{\gamma^{\pm}(\bar{r}) \mid \bar{r} \in \bar{R}\}$$

The domain of maximal set of concrete conditional traces and the domain of maximal sets of abstract conditional traces are related by a Galois Insertion as stated by the following lemma.

Lemma 4.2.8 *The pair of functions* $(\alpha^{\pm}, \gamma^{\pm})$ *is a Galois Insertion*

$$(\mathbf{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}, \{\epsilon\}) \underset{\alpha^{\pm}}{\overset{\gamma^{\pm}}{\leftrightarrows}} (\mathbf{M}^{\pm}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}^{\pm}, \{\epsilon\})$$

4.2.3 Abstraction of the conditional traces structure

Abstracting the information contained in the traces, often leads to abstract traces that present equal consecutive conditional states. For instance, consider the concrete trace $r = (x > 0, \varnothing) \rightarrowtail x > 0 \cdot (x > 2, \varnothing) \rightarrowtail x > 2 \cdot (x > 5, \varnothing) \rightarrowtail x > 5 \cdot \boxtimes$ together with the positive/negative approximations defined in Example 4.2.2, the result is the trace $\alpha^{\pm}(r) = (\mathrm{pos}_x, \varnothing) \rightarrowtail \mathrm{pos}_x \cdot (\mathrm{pos}_x, \varnothing) \rightarrowtail \mathrm{pos}_x \cdot (\mathrm{pos}_x, \varnothing) \rightarrowtail \mathrm{pos}_x \cdot \boxtimes$. Therefore, we can think on collapse these consecutive states in an unique one to obtain a more compact representation, for instance a trace on the form $(\mathrm{pos}_x, \varnothing) \rightarrowtail \mathrm{pos}_x \cdot \boxtimes$. However, due to the particularly strong synchronization notion of the language, this information is not enough for our verification and analysis purposes. As already noticed in [4], the loss of synchronization in other *ccp* languages just implies a loss of precision, but in the case of *tccp*, due to the maximal parallelism, it would imply a loss of correctness. Thus, we need to know how long each *fragment* of the computation is. The idea to solve this problem is to associate a natural number to each abstract conditional tuple. This allows us to keep synchronization among processes. For instance in the previous example we obtain an abstract trace of the form $[(\mathrm{pos}_x, \varnothing) \rightarrowtail \mathrm{pos}_x]^3 \cdot \boxtimes$.

This observation lead us to the notion of *compact abstract conditional trace*:

Definition 4.2.9 (Compact abstract conditional trace) *A* compact abstract conditional trace *(briefly compact abstract trace) is a trace, possibly ended with $\boxtimes$, of abstract states with a natural number m_i associated: $\tilde{t}_1^{m_1} \dots \tilde{t}_n^{m_n} \dots$, such that for each $\tilde{t}_i^{m_i}$ and $\tilde{t}_j^{m_j}$ with $i > j$, $\tilde{t}_i \neq \tilde{t}_j$, i.e., two consecutive states must be different. In addition, a compact abstract trace satisfies the properties of monotonicity and consistency in Definition 4.2.6 must hold.*

It can be noticed that, if the associated number is 0, the tuple corresponds to the empty sequence ϵ. Furthermore, to obtain always a well formed compact abstract conditional trace, we assume that the concatenation of two equal abstract conditional states such as $[(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{c}]^n \cdot [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{c}]^m$ is interpreted as the single abstract conditional state $[(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{c}]^{n+m}$.

Let $\tilde{r}$, $\tilde{r}_1$ and $\tilde{r}_2$ be compact abstract traces and $\tilde{t}^n$ a state with a natural number associated, we define the ordering of compact abstract traces as

$$\epsilon < \boxtimes \leq \tilde{r} \quad \forall \tilde{r} \neq \epsilon$$
$$\tilde{t}^n \cdot \tilde{r}_1 \leq \tilde{t}^n \cdot \tilde{r}_2 \iff \tilde{r}_1 \leq \tilde{r}_2$$
$$\tilde{t}^n \leq \tilde{t}^m \cdot \tilde{r} \iff n < m$$

We denote by $\tilde{\mathbf{M}}^{\pm}$ the set of all compact abstract traces.
It is easy to see that $(\tilde{\mathbf{M}}^{\pm}, \leq, \bigvee, \bigwedge, \tilde{\mathbf{M}}^{\pm}, \{\epsilon\})$ is a complete lattice, where the relation $\leq$ is lifted to sets of traces in the following way: given $\tilde{R}_1, \tilde{R}_2 \in \tilde{\mathbf{M}}^{\pm}$, $\tilde{R}_1 \leq \tilde{R}_2 \iff \forall \tilde{r}_1 \in \tilde{R}_1\ \exists \tilde{r}_2 \in \tilde{R}_2$ such that $\tilde{r}_1 \leq \tilde{r}_2$.

In the following, we state formally the relation between the abstract traces of Definition 4.2.6 and the compact ones. To this end, we need to define two auxiliary functions: the function κ computes a compact abstract trace from an abstract one, whereas the function κ^{-1} gets the correspondent abstract trace from a compact one.

Definition 4.2.10 *The function $\kappa\colon \mathbf{M}^{\pm} \mapsto \tilde{\mathbf{M}}^{\pm}$, given an abstract trace $r \in \mathbf{M}^{\pm}$, collapses all its equal consecutive abstract conditional states in the following way:*

$$\kappa(\epsilon) := \epsilon \tag{4.2.14}$$

$$\kappa(\boxtimes) := \boxtimes \tag{4.2.15}$$

$$\kappa(\tilde{\eta} \rightarrowtail \hat{c} \cdot \bar{r}) := \begin{cases} [\tilde{\eta} \rightarrowtail \hat{c}]^{n+1} \cdot \tilde{r}' & \textit{if } \kappa(r) = [\tilde{\eta} \rightarrowtail \hat{c}]^{n} \cdot \tilde{r}' \textit{ and } n > 0 \\ [\tilde{\eta} \rightarrowtail \hat{c}]^{1} \cdot \kappa(r) & \textit{otherwise} \end{cases} \tag{4.2.16}$$

$$\kappa(\mathit{stutt}(\breve{\eta}) \cdot r) := \begin{cases} [\mathit{stutt}(\breve{\eta})]^{n+1} \cdot \tilde{r}' & \textit{if } \kappa(\bar{r}) = [\mathit{stutt}(\breve{\eta})]^{n} \cdot \tilde{r}' \textit{ and } n > 0 \\ [\mathit{stutt}(\breve{\eta})]^{1} \cdot \kappa(r) & \textit{otherwise} \end{cases} \tag{4.2.17}$$

The function $\kappa^{-1}\colon \tilde{\mathbf{M}}^{\pm} \mapsto \mathbf{M}^{\pm}$, given a compact abstract conditional trace $r \in \tilde{\mathbf{M}}^{\pm}$, expands each abstract conditional state according to its associated number:

$$\kappa^{-1}(\epsilon) := \epsilon \tag{4.2.18}$$

$$\kappa^{-1}(\boxtimes) := \boxtimes \tag{4.2.19}$$

$$\kappa^{-1}([\tilde{\eta} \rightarrowtail \hat{c}]^{n} \cdot \tilde{r}) := \underbrace{\tilde{\eta} \rightarrowtail \hat{c} \ldots \tilde{\eta} \rightarrowtail \hat{c}}_{n \textit{ times}} \cdot \kappa^{-1}(\tilde{r}) \tag{4.2.20}$$

$$\kappa^{-1}([\mathit{stutt}(\breve{\eta})]^{n} \cdot \tilde{r}) := \underbrace{\mathit{stutt}(\breve{\eta}) \ldots \mathit{stutt}(\breve{\eta})}_{n \textit{ times}} \cdot \kappa^{-1}(\tilde{r}) \tag{4.2.21}$$

It is worth noticing that $\kappa \circ \kappa^{-1} = \kappa^{-1} \circ \kappa = id$ and κ and κ^{-1} are idempotent.

We abuse in notation by denoting as (κ, κ^{-1}) the pair of functions that relate sets of abstract traces to sets of compact abstract traces in the following way:

$$\kappa(R) = \bigvee\{\kappa(r) \mid r \in R\}$$
$$\kappa^{-1}(\tilde{r}) = \bigsqcup\{\kappa^{-1}(\tilde{r}) \mid \tilde{r} \in \tilde{r}\}$$

Lemma 4.2.11 *The pair of functions (κ, κ^{-1}) is an order-preserving isomorphism*

$$(\mathbf{M}^{\pm}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}^{\pm}, \{\epsilon\}) \overset{\kappa^{-1}}{\underset{\kappa}{\leftrightarrows}} (\tilde{\mathbf{M}}^{\pm}, \leq, \bigvee, \bigwedge, \tilde{\mathbf{M}}^{\pm}, \{\epsilon\})$$

By composition of the Galois Insertions $(\alpha^{\pm}, \gamma^{\pm})$ and the isomorphism (κ, κ^{-1}) we obtain

$$(\mathbf{M}, \sqsubseteq) \overset{\gamma^{\pm}}{\underset{\alpha^{\pm}}{\leftrightarrows}} (\mathbf{M}^{\pm}, \sqsubseteq) \overset{\kappa^{-1}}{\underset{\kappa}{\leftrightarrows}} (\tilde{\mathbf{M}}^{\pm}, \leq) \tag{4.2.22}$$

We denote this compositions as $\tilde{\alpha} = \alpha^{\pm} \circ \kappa$ and $\tilde{\gamma} = \kappa^{-1} \circ \gamma^{\pm}$, obtaining the following Galois Insertion:

$$(\mathbf{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}, \{\epsilon\}) \overset{\tilde{\gamma}}{\underset{\tilde{\alpha}}{\leftrightarrows}} (\tilde{\mathbf{M}}^{\pm}, \leq, \bigvee, \bigwedge, \tilde{\mathbf{M}}^{\pm}, \{\epsilon\}) \tag{4.2.23}$$

This abstraction can be systematically lift to the domain of interpretations: $\mathbf{I} \overset{\tilde{\gamma}}{\underset{\tilde{\alpha}}{\leftrightarrows}} [\mathbf{PC} \to \tilde{\mathbf{M}}^{\pm}]$. Elements of $\mathbf{I}^{\pm} := [\mathbf{PC} \to \tilde{\mathbf{M}}^{\pm}]$ are called abstract interpretations.

Let us show some examples of the application of the abstraction function $\tilde{\alpha}$.

Example 4.2.12

We apply the abstraction function $\tilde{\alpha}$ to the fixpoint semantics $\mathcal{F}[\![D]\!]$ computed in Example 3.1.22 by using the abstract constraint system $\mathbf{F\hat{D}}(5)$ (Example 4.2.1). Let us consider, first, the concrete trace $\bar{r} := (x = 4, \varnothing) \rightarrowtail x = 4 \cdot (x = 4, \varnothing) \rightarrowtail x = 4$. By applying to $\bar{r}$ the abstraction function $\alpha^{\pm}$ it is easy to see that we obtain the following abstract trace $\alpha^{\pm}(\bar{r}) := (x \hat{=} 4, \varnothing) \rightarrowtail x \hat{=} 4 \cdot (x \hat{=} 4, \varnothing) \rightarrowtail x \hat{=} 4$, and, by further applying κ we obtain the compact abstract trace $\kappa(\alpha^{\pm}(\bar{r})) := [(x \hat{=} 4, \varnothing) \rightarrowtail x \hat{=} 4]^2$. Now, we apply $\tilde{\alpha}$ to the result of the fixpoint semantics $\mathcal{F}[\![D]\!]$.

$$
\begin{aligned}
&\tilde{\alpha}(\mathcal{F}[\![D]\!]) \\
&= \begin{cases} p(x) \mapsto \kappa(\alpha^{\pm}(\, \{(\mathit{stutt}(\{x = 4\}))^n \cdot \bar{r} \cdots \bar{r} \cdots \mid n \in \mathbf{N}\} \sqcup \,)) \\ \qquad\qquad \{\mathit{stutt}(\{x = 4\}) \cdots \mathit{stutt}(\{x = 4\}) \cdots\} \end{cases} \\
&= \begin{cases} p(x) \mapsto \{\kappa(\alpha^{\pm}((\mathit{stutt}(\{x = 4\}))^n \cdot \bar{r} \cdots \bar{r} \cdots)) \mid n \in \mathbf{N}\} \vee \\ \qquad\quad \{\kappa(\alpha^{\pm}(\mathit{stutt}(\{x = 4\}) \cdots \mathit{stutt}(\{x = 4\}) \cdots))\} \end{cases} \\
&\qquad [\text{ by Definition 4.2.7 }] \\
&= \begin{cases} p(x) \mapsto \{\kappa((\mathit{stutt}(\tau^-(\{x = 4\})))^n \cdot \\ \qquad\quad (\tau^+(x = 4), \tau^-(\varnothing)) \rightarrowtail \tau^+(x = 4) \cdots (\tau^+(x = 4), \tau^-(\varnothing)) \rightarrowtail \tau^+(x = 4) \cdots) \mid n \in \mathbf{N}\} \vee \\ \qquad\quad \{\kappa(\mathit{stutt}(\tau^-(\{x = 4\})) \cdots \mathit{stutt}(\tau^-(\{x = 4\})) \cdots)\} \end{cases} \\
&\qquad [\text{ by Definitions of } \tau^+ \text{ and } \tau^- \text{ in Example 4.2.1 }] \\
&= \begin{cases} p(x) \mapsto \{\kappa((\mathit{stutt}(\{x \hat{=} 4\}))^n \cdot (x \hat{=} 4, \varnothing) \rightarrowtail x \hat{=} 4 \cdots (x \hat{=} 4, \varnothing) \rightarrowtail x \hat{=} 4 \cdots) \mid n \in \mathbf{N}\} \vee \\ \qquad\quad \{\kappa(\mathit{stutt}(\{x \hat{=} 4\}) \cdots \mathit{stutt}(\{x \hat{=} 4\}) \cdots)\} \end{cases} \\
&\qquad [\text{ by Definition of } \kappa \text{ (4.2.14) }] \\
&= \begin{cases} p(x) \mapsto \{[\mathit{stutt}(\{x \hat{=} 4\})]^n \cdot [(x \hat{=} 4, \varnothing) \rightarrowtail x \hat{=} 4]^{+\infty} \mid n \in \mathbf{N}\} \vee \\ \qquad\quad \{[\mathit{stutt}(\{x \hat{=} 4\})]^{+\infty}\} \end{cases}
\end{aligned}
$$

The abstract semantics shows that the procedure $p(x)$ loops on the store $x \hat{=} 4$ if the guard $x \hat{=} 4$ is entailed by the current store, otherwise it waits for a finite or infinite time for the guard to be entailed and eventually loops on the store $x \hat{=} 4$. Notice that the function κ allows us to effectively collapse the equal consecutive states and obtain a compact representation for the abstract traces.

We show a second abstraction example for the semantics.

Example 4.2.13

We apply the abstraction function $\tilde{\alpha}$ to the fixpoint semantics $\mathcal{F}[\![D]\!]$ computed in Example 3.1.21 by using the abstract constraint system defined in 4.2.2.

$$
\tilde{\alpha}(\mathcal{F}[\![D]\!]) = \begin{cases} \{q(x, y) \mapsto \bigvee\{[(\hat{\mathit{true}}, \{\mathrm{pos}_x\}) \rightarrowtail \hat{\mathit{true}}]^n \cdot [(\mathrm{pos}_x, \varnothing) \rightarrowtail \mathrm{pos}_x \hat{\otimes} \mathrm{neg}_y]^1 \cdot \boxtimes \mid n \in \mathbf{N}\} \\ \qquad\qquad\qquad \vee\{[(\hat{\mathit{true}}, \{\mathrm{pos}_x\}) \rightarrowtail \hat{\mathit{true}}]^{+\infty}\} \end{cases}
$$

Notice that, due to the abstraction of the constraint system, we lose the real content of the global store, namely we do not have the information that $x > 2$ and $y < 0$ anymore.

4.3 Induced Abstract Semantics

In general, it is not possible to compute the fixpoint semantics $\mathcal{F}$ in finite time. Thus, given a set of declarations D we cannot just compute $\tilde{\alpha}(\mathcal{F}[\![D]\!])$ to to obtain the abstract semantics of D.

For this reason, in this section, we present an abstract semantics for *tccp*, which is obtained by abstracting the small-step semantics of Section 3.1 with the abstraction framework defined in Section 4.2. Since we will use standard abstract interpretation results, the obtained semantics turns out to be the best correct approximation of $\mathcal{F}$ in the domain of compact abstract conditional traces.

Let us introduce, first, the abstract counterpart of the concrete auxiliary semantic functions introduced in Section 3.1.2. We show that these abstract functions are a correct approximation of the concrete ones. This property is called local correctness of the abstract operators and it is necessary to show that the final abstract semantics (Theorem 4.A.1) is (globally) correct.

The *abstract propagation operator* $\tilde{\downarrow}$ is a partial function $\tilde{\mathbf{M}}^{\pm} \times \hat{\mathbf{C}} \to \tilde{\mathbf{M}}^{\pm}$ which propagates the information of an abstract constraint in an abstract trace and checks for the consistency of the new information with the conditional states in the abstract trace.

Definition 4.3.1 (Abstract propagation operator) *Let $\tilde{r} \in \tilde{\mathbf{M}}^{\pm}$ and $\hat{c} \in \hat{\mathbf{C}}$. We define the abstract propagation of $\hat{c}$ in $\tilde{r}$, written $\tilde{r}\tilde{\downarrow}_{\hat{c}}$, as $\boxtimes\tilde{\downarrow}_{\hat{c}} = \boxtimes$, $\epsilon\tilde{\downarrow}_{\hat{c}} = \epsilon$ and*

$$([(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}]^n \cdot \tilde{r}')\tilde{\downarrow}_{\hat{c}} = \begin{cases} [(\hat{c} \hat{\otimes} \hat{\eta}, \check{\eta}) \rightarrowtail \hat{c} \hat{\otimes} \hat{a}]^n \cdot (\tilde{r}'\tilde{\downarrow}_{\hat{c}}) & \text{if } \hat{c} \tilde{\gg} (\hat{\eta}, \check{\eta}),\ \hat{c} \hat{\otimes} \hat{d} \neq \widehat{false} \\ [(\hat{c} \hat{\otimes} \hat{\eta}, \check{\eta}) \rightarrowtail \widehat{false}]^1 & \text{if } \hat{c} \tilde{\gg} (\hat{\eta}, \check{\eta}),\ \hat{c} \hat{\otimes} \hat{d} = \widehat{false} \end{cases}$$

$$([stutt(\check{\eta})]^n \cdot \tilde{r}')\tilde{\downarrow}_{\hat{c}} = [stutt(\check{\eta})]^n \cdot (\tilde{r}'\tilde{\downarrow}_{\hat{c}}) \qquad \text{if } \hat{c} \tilde{\not\vdash} \check{\eta}$$

The abstract propagation operator is correct w.r.t. the concrete one, as formally stated by the following lemma.

Lemma 4.3.2 *Let $r \in \mathbf{M}$ and $c \in \mathbf{C}$, then $\tilde{\alpha}(r\downarrow_c) = \tilde{\alpha}(r)\tilde{\downarrow}_{\tau^+(c)}$.*

The following definition extends the notion of compatibility (Definition 3.1.9) to abstract traces.

Definition 4.3.3 ($\hat{c}$-compatible) *$\tilde{r} \in \tilde{\mathbf{M}}^{\pm}$ is said to be* abstractly compatible *w.r.t. $\hat{c} \in \hat{\mathbf{C}}$ ($\tilde{r}$ is $\hat{c}$-compatible) if, for each $[(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}]^n$ in $\tilde{r}$, $\hat{c} \tilde{\gg} (\hat{\eta}, \check{\eta})$, and for each $[stutt(\check{\eta})]^n$ in $\tilde{r}$, $\hat{c} \tilde{\not\vdash} \check{\eta}$.*

A trace $\tilde{r}$ is not $\hat{c}$-compatible when $\hat{c}$ is in contradiction with a condition in $\tilde{r}$, in this case $\hat{c}\tilde{\downarrow}_{\tilde{r}}$ is not defined.

The following lemma follows directly from condition (4.2.1i) and Equation (4.2.3).

Proposition 4.3.4 *Given $r \in \mathbf{M}$ and $c \in \mathbf{C}$,*

$$r \text{ is } c\text{-compatible} \Longrightarrow \tilde{\alpha}(r) \text{ is abstractly } \tau^+(c)\text{-compatible.}$$

The *abstract parallel composition operator* combines two abstract conditional traces in terms of maximal parallelism. As its concrete counterpart (Definition 3.1.10), it checks the satisfiability of the conditions and the consistency of the resulting stores.

Definition 4.3.5 (Abstract parallel composition) *The* abstract parallel composition *partial operator* $\tilde{\|} : \tilde{\mathbf{M}}^{\pm} \times \tilde{\mathbf{M}}^{\pm} \to \tilde{\mathbf{M}}^{\pm}$ *is the commutative closure of the following partial operation defined by structural induction as:* $\tilde{r} \mathbin{\tilde{\|}} \epsilon := \tilde{r}$, $\tilde{r} \mathbin{\bar{\|}} \boxtimes := r$ *and, if* $n \leq m$,

$$([\mathit{stutt}(\check{\eta}_1)]^n \cdot \tilde{r}'_1) \mathbin{\tilde{\|}} ([\mathit{stutt}(\check{\eta}_2)]^m \cdot \tilde{r}'_2) := [\mathit{stutt}(\check{\eta}_1 \mathbin{\check{\oplus}} \check{\eta}_2)]^n \cdot (\tilde{r}'_1 \mathbin{\tilde{\|}} [\mathit{stutt}(\check{\eta}_2)]^{m-n} \cdot \tilde{r}'_2)$$

Moreover, if $\tilde{\eta}_1 \mathbin{\tilde{\otimes}} \tilde{\eta}_2$ *is valid,* $\tilde{r}'_1$ *is* $\hat{c}_2$*-compatible,* $\tilde{r}'_2$ *is* $\hat{c}_1$*-compatible and* $n \leq m$, *then*

$$([\tilde{\eta}_1 \rightarrowtail \hat{c}_1]^n \cdot \tilde{r}'_1) \mathbin{\tilde{\|}} ([\hat{\eta}_2 \rightarrowtail \hat{c}_2]^m \cdot \tilde{r}'_2) :=$$
$$\begin{cases} [\hat{\eta}_1 \mathbin{\tilde{\otimes}} \hat{\eta}_2 \rightarrowtail \hat{c}_1 \otimes \hat{c}_2]^n \cdot & \\ \qquad ((r'_1 \tilde{\downarrow}_{\hat{c}_2}) \mathbin{\bar{\|}} ((([\hat{\eta}_2 \rightarrowtail \hat{c}_2]^{m-n} \cdot r'_2) \tilde{\downarrow}_{\hat{c}_1})) & \text{if } \hat{c}_1 \mathbin{\hat{\otimes}} \hat{c}_2 \neq \widehat{\mathit{false}} \\ [\hat{\eta}_1 \mathbin{\tilde{\otimes}} \hat{\eta}_2 \rightarrowtail \widehat{\mathit{false}}]^1 \cdot \boxtimes & \text{if } \hat{c}_1 \mathbin{\hat{\otimes}} \hat{c}_2 = \widehat{\mathit{false}} \end{cases}$$

Finally, if $\hat{\eta}_1^+ \mathrel{\tilde{\nVdash}} \check{\eta}_2$ *and* $\tilde{r}'_2$ *is* $\hat{c}_1$*-compatible, then*

$$([\tilde{\eta}_1 \rightarrowtail \hat{c}_1]^n \cdot \tilde{r}'_1) \mathbin{\tilde{\|}} ([\mathit{stutt}(\check{\eta}_2)]^m \cdot \tilde{r}'_2) :=$$
$$\begin{cases} [(\hat{\eta}_1, \check{\eta}_1 \mathbin{\check{\oplus}} \check{\eta}_2) \rightarrowtail \hat{c}_1]^n \cdot (\tilde{r}'_1 \mathbin{\bar{\|}} ((([\mathit{stutt}(\check{\eta}_2)]^{m-n} \cdot \tilde{r}'_2) \tilde{\downarrow}_{\hat{c}_1})) & \text{if } n \leq m \\ [(\hat{\eta}_1, \check{\eta}_1 \mathbin{\check{\oplus}} \check{\eta}_2) \rightarrowtail \hat{c}_1]^m \cdot ([\tilde{\eta}_1 \rightarrowtail \hat{c}_1]^{n-m} \cdot \tilde{r}'_1 \mathbin{\bar{\|}} (\tilde{r}'_2 \tilde{\downarrow}_{\hat{c}_1})) & \text{if } m < n \end{cases}$$

Similarly to the concrete case, $\tilde{\|}$ is commutative and associative and $\tilde{\downarrow}$ distributes over $\tilde{\|}$ (i.e., $(\tilde{r}_1 \mathbin{\tilde{\|}} \tilde{r}_2) \tilde{\downarrow}_{\hat{c}} = (\tilde{r}_1 \tilde{\downarrow}_{\hat{c}}) \mathbin{\tilde{\|}} (\tilde{r}_2 \tilde{\downarrow}_{\hat{c}})$). It is worth noting that, if one of the traces is not compatible with the propagated abstract constraint, the abstract parallel composition is not defined.

The following lemma states the soundness of $\tilde{\|}$ w.r.t. the concrete parallel composition operator (Definition 3.1.10).

Lemma 4.3.6 *Let* $r_1, r_2 \in \mathbf{M}$, $\tilde{\alpha}(r_1 \mathbin{\bar{\|}} r_2) = \tilde{\alpha}(r_1) \mathbin{\tilde{\|}} \tilde{\alpha}(r_2)$ *holds.*

The *abstract hiding operator* $\tilde{\exists} : \mathcal{V} \times \tilde{\mathbf{M}}^{\pm} \to \tilde{\mathbf{M}}^{\pm}$ hides the information regarding a given variable in an abstract conditional trace.

Definition 4.3.7 (Abstract hiding operator) *Given* $\tilde{r} \in \tilde{\mathbf{M}}^{\pm}$ *and* $x \in \mathcal{V}$, *we define the* hiding of x in $\tilde{r}$, *written* $\tilde{\exists}_x \tilde{r}$, *by structural induction:*

$$\tilde{\exists}_x \tilde{r} := \begin{cases} [(\hat{\exists}_x \eta^+, \check{\exists}_x \eta^-) \rightarrowtail \hat{\exists}_x \hat{a}]^n \cdot \tilde{\exists}_x \tilde{r}' & \text{if } \tilde{r} = [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{a}]^n \cdot \tilde{r}' \\ [\mathit{stutt}(\check{\exists}_x \check{\eta})]^n \cdot \tilde{\exists}_x \tilde{r}' & \text{if } \tilde{r} = [\mathit{stutt}(\check{\eta})]^n \cdot \tilde{r}' \\ \tilde{r} & \text{if } \tilde{r} = \epsilon \text{ or } \tilde{r} = \boxtimes \end{cases}$$

The abstract hiding operator $\tilde{\exists}$ is sound w.r.t. its concrete counterpart (Definition 3.1.12).

Lemma 4.3.8 *Given* $r \in \mathbf{M}$ *and* $x \in \mathit{Var}$, $\tilde{\alpha}(\bar{\exists}_x r) = \kappa(\tilde{\exists}_x \tilde{\alpha}(r))$.

As in the concrete case, we distinguish two special classes of abstract conditional traces.

Definition 4.3.9 (Abstractly Self-sufficient and x-self-sufficient conditional trace) *An abstract trace $\tilde{r} \in \tilde{\mathbf{M}}^{\pm}$ is said to be* abstractly self-sufficient *if the first condition is* $(\hat{true}, \check{false})$ *and, for each* $\tilde{t}_i = [(\hat{\eta}_i, \check{\eta}_i) \rightarrowtail \hat{c}_i]^n$ *and* $\tilde{t}_{i+1} = [(\hat{\eta}_{i+1}, \check{\eta}_{i+1}) \rightarrowtail \hat{c}_{i+1}]^n$, $\hat{c}_i \Vdash\!\!\tilde{=} \eta_{i+1}$. *In other words, each abstract store (abstractly) satisfies the successive abstract condition.*

Moreover, $\tilde{r}$ *is* abstractly self-sufficient w.r.t. $x \in \mathcal{V}$ *(*x-self-sufficient*) if* $\tilde{\exists}_{Var \smallsetminus \{x\}} \tilde{r}$ *is self-sufficient.*

It follows directly from Lemma 4.3.8 and Equation (4.2.13) that, given a conditional trace $r \in \mathbf{M}$ and a variable $x \in Var$:

$$r \text{ is } x\text{-self-sufficient} \implies \tilde{\alpha}(r) \text{ is abstractly } x\text{-self-sufficient} \tag{4.3.1}$$

Now that we have introduced all the essential auxiliary operator, we can derive the optimal abstract version $\mathcal{D}^{\pm}[\![D]\!]$ of $\mathcal{D}[\![D]\!]$ simply as

$$\mathcal{D}^{\pm}[\![D]\!] := \tilde{\alpha} \circ \mathcal{D}[\![D]\!] \circ \tilde{\gamma}$$

It turns out (Theorem 4.A.1) that for a given abstract interpretation $\mathcal{I}^{\pm}$:

$$\mathcal{D}^{\pm}[\![D]\!]_{\mathcal{I}^{\pm}} = \lambda p(x). \bigvee\nolimits_{p(x) :- A \in D} \mathcal{A}^{\pm}[\![A]\!]_{\mathcal{I}^{\pm}} \tag{4.3.2}$$

where $\mathcal{A}^{\pm}$ is defined by structural induction on the syntax in a similar way as the concrete version.

$$\mathcal{A}^{\pm}[\![\mathsf{skip}]\!]_{\mathcal{I}^{\pm}} := \boxtimes \tag{4.3.3}$$

$$\mathcal{A}^{\pm}[\![\mathsf{tell}(c)]\!]_{\mathcal{I}^{\pm}} := [(\hat{true}, \check{false}) \rightarrowtail \tau^{+}(c)]^1 \cdot \boxtimes \tag{4.3.4}$$

$$\mathcal{A}^{\pm}[\![A \parallel B]\!]_{\mathcal{I}^{\pm}} := \bigvee \big\{ \tilde{r}_A \mathbin{\tilde{\parallel}} \tilde{r}_B \mid \tilde{r}_A \in \mathcal{A}^{\pm}[\![A]\!]_{\mathcal{I}^{\pm}}, \tilde{r}_B \in \mathcal{A}^{\pm}[\![B]\!]_{\mathcal{I}^{\pm}} \big\} \tag{4.3.5}$$

$$\mathcal{A}^{\pm}[\![\exists x\, A]\!]_{\mathcal{I}^{\pm}} := \bigvee \big\{ \tilde{\exists}_x \tilde{r} \mid \tilde{r} \in \mathcal{A}^{\pm}[\![A]\!]_{\mathcal{I}^{\pm}}, \tilde{r} \text{ is abstracly } x\text{-self-sufficient} \big\} \tag{4.3.6}$$

$$\mathcal{A}^{\pm}[\![p(z)]\!]_{\mathcal{I}^{\pm}} := \bigvee \big\{ [(\hat{true}, \check{false}) \rightarrowtail \hat{true}]^1 \cdot \tilde{r} \mid \tilde{r} \in \mathcal{I}^{\pm}(p(z)) \big\} \tag{4.3.7}$$

$$\mathcal{A}^{\pm}[\![\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}^{\pm}} := lfp_{\tilde{\mathbf{M}}^{\pm}}\, \lambda \tilde{R}. \Big(([stutt(\tau^{-}(\{c_1, \dots, c_n\}))]^1 \cdot \tilde{R}) \vee \tag{4.3.8}$$

$$\bigvee \big\{ [(\tau^{+}(c_i), \check{false}) \rightarrowtail \tau^{+}(c_i)]^1 \cdot (\tilde{r} \mathbin{\tilde{\downarrow}}_{\tau^{+}(c_i)}) \,\big|\, 1 \le i \le n,\ \tilde{r} \in \mathcal{A}^{\pm}[\![A_i]\!]_{\mathcal{I}^{\pm}},\ \tilde{r}\ \tau^{+}(c_i)\text{-compatible} \big\} \Big) \tag{4.3.9}$$

$$\mathcal{A}^{\pm}[\![\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\mathcal{I}^{\pm}} :=$$

$$\{ [(\tau^{+}(c), \check{false}) \rightarrowtail \tau^{+}(c)]^1 \cdot \boxtimes \mid \boxtimes \in \mathcal{A}^{\pm}[\![A]\!]_{\mathcal{I}^{\pm}} \} \vee \tag{4.3.10}$$

$$\bigvee \{ [(c \mathbin{\hat{\times}} \hat{\eta}, \check{\eta}) \rightarrowtail c \mathbin{\hat{\times}} \hat{d}]^n \cdot (\tilde{r} \mathbin{\tilde{\downarrow}}_{\tau^{+}(c)}) \mid [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}]^n \cdot \tilde{r} \in \mathcal{A}^{\pm}[\![A]\!]_{\mathcal{I}^{\pm}}, \tag{4.3.11}$$

$$c \mathbin{\hat{\times}} \hat{d} \neq \hat{false}\ ,\ c \mathbin{\hat{\times}} \hat{\eta} \mathrel{\tilde{\nVdash}} \check{\eta},\ \tilde{r}\ \tau^{+}(c)\text{-compatible} \} \sqcup$$

$$\bigvee \{ [(c \mathbin{\hat{\times}} \hat{\eta}, \check{\eta}) \rightarrowtail \hat{false}]^1 \} \cdot \boxtimes \mid [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}]^n \cdot \tilde{r} \in \mathcal{A}^{\pm}[\![A]\!]_{\mathcal{I}^{\pm}}, \tag{4.3.12}$$

$$c \mathbin{\hat{\times}} \hat{d} = \hat{false},\ c \mathbin{\hat{\times}} \hat{\eta} \mathrel{\tilde{\nVdash}} \check{\eta},\ \tilde{r}\ \tau^{+}(c)\text{-compatible} \} \vee$$

$$\bigvee \{ [(\tau^{+}(c), \check{\eta}) \rightarrowtail \tau^{+}(c)]^1 \cdot [stutt(\check{\eta})]^n \cdot \kappa(\tilde{r} \mathbin{\tilde{\downarrow}}_{\tau^{+}(c)}) \mid \tag{4.3.13}$$

$$[stutt(\check{\eta})]^{n+1} \cdot \tilde{r} \in \mathcal{A}^{\pm}[\![A]\!]_{\mathcal{I}^{\pm}},\ \tau^{+}(c) \mathrel{\tilde{\nVdash}} \check{\eta},\ \tilde{r}\ \tau^{+}(c)\text{-compatible} \} \vee$$

$$\bigvee \{ [(\hat{true}, \tau^{-}(c)) \rightarrowtail \hat{true}]^1 \cdot \boxtimes \mid \boxtimes \in \mathcal{A}^{\pm}[\![B]\!]_{\mathcal{I}^{\pm}} \} \vee \tag{4.3.14}$$

$$\bigvee\{[(\hat{\eta}, c \check{\times} \check{\eta}) \rightarrowtail \hat{a}]^n \cdot \tilde{r} \mid [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{a}]^n \cdot \tilde{r} \in \mathcal{A}^{\pm}[\![B]\!]_{\mathcal{I}^{\pm}},\, \hat{\eta} \mathrel{\hat{\nvdash}} \tau^{-}(\{c\})\} \vee \tag{4.3.15}$$

$$\bigvee\{[(\widehat{true}, c \check{\times} \check{\eta}) \rightarrowtail \widehat{true}]^1 \cdot [stutt(\check{\eta})]^n \cdot \tilde{r} \mid [stutt(\check{\eta})]^{n+1} \cdot \tilde{r} \in \mathcal{A}^{\pm}[\![B]\!]_{\mathcal{I}^{\pm}}\} \tag{4.3.16}$$

Abstract interpretation theory assures that $\mathcal{F}^{\pm}[\![D]\!] := \mathit{lfp}\,\mathcal{D}^{\pm}[\![D]\!]$ is the best correct approximation of $\mathcal{F}[\![D]\!]$.[2]

Let us remark that the considered domain is not noetherian. In fact, it is easy to build infinite ascending chains as $\{[\tilde{\eta} \rightarrowtail \hat{a}]^1\} \subseteq \cdots \subseteq \{[\tilde{\eta} \rightarrowtail \hat{a}]^1, \ldots, [\tilde{\eta} \rightarrowtail \hat{a}]^n\} \cdots \subseteq \{[\tilde{\eta} \rightarrowtail \hat{a}]^1, \ldots, [\tilde{\eta} \rightarrowtail \hat{a}]^{+\infty}\} \ldots$. For this reason, we can not guarantee that the fixed point can be reached in a *finite* number of steps. However, starting from a finite (abstract) interpretation, each single iteration terminates in finite time. This makes the semantics suitable for abstract diagnosis (see Section 4.4).

Let us show some examples of the calculus of the abstract semantics $\mathcal{F}^{\pm}$.

Example 4.3.10

Consider the set of declaration of Example 3.1.22: $D := \{p(x) \coloneq \mathsf{ask}(x = 4) \rightarrow p(x)\}$.

The abstract semantics of the agent $\mathsf{ask}(x = 4) \rightarrow p(x)$ by using the abstraction over constraints $\mathbf{F\hat{D}}(5)$ defined in Example 4.2.1 is computed as follows.

$$\begin{aligned}
&\mathcal{A}^{\pm}[\![\mathsf{ask}(x = 4) \rightarrow p(x)]\!]_{\mathcal{I}^{\pm}} \\
&\quad = \mathit{lfp}_{\tilde{\mathbf{M}}^{\pm}}\, \lambda \tilde{R}. \Big(([stutt(\tau^{-}(\{x = 4\}))]^1 \cdot \tilde{R}) \vee \\
&\qquad\qquad \bigvee\big\{[(\tau^{+}(x = 4), \tau^{-}(\varnothing)) \rightarrowtail \tau^{+}(x = 4)]^1 \cdot (\tilde{r}\tilde{\downarrow}_{\tau^{+}(x=4)}) \,\big|\, \tilde{r} \in \mathcal{A}^{\pm}[\![p(x)]\!]_{\mathcal{I}^{\pm}}\big\}\Big) \\
&\quad = \mathit{lfp}_{\tilde{\mathbf{M}}^{\pm}}\, \lambda \tilde{R}. \Big(([stutt(\{x \mathrel{\hat{=}} 4\})]^1 \cdot \tilde{R}) \vee \\
&\qquad \bigvee\big\{[(x \mathrel{\hat{=}} 4, \varnothing) \rightarrowtail x \mathrel{\hat{=}} 4]^1 \cdot (\tilde{r}\tilde{\downarrow}_{x \hat{=} 4}) \,\big|\, \tilde{r} \in \mathcal{A}^{\pm}[\![p(x)]\!]_{\mathcal{I}^{\pm}}\big\}\Big) \\
&\quad = \big\{[stutt(\{x \mathrel{\hat{=}} 4\})]^{+\infty}\big\} \vee \\
&\qquad \bigvee\big\{[stutt(\{x \mathrel{\hat{=}} 4\})]^n \cdot [(x \mathrel{\hat{=}} 4, \varnothing) \rightarrowtail x \mathrel{\hat{=}} 4]^2 \cdot \tilde{r} \,\big|\, \tilde{r} \in \mathcal{I}^{\pm}(p(x)), n \in \mathbf{N}\big\}
\end{aligned}$$

The iterates of $\mathcal{D}^{\pm}[\![D]\!]$ are

$$\mathcal{D}^{\pm}[\![D]\!]{\uparrow}1 = \begin{cases} p(x) \mapsto & \{[stutt(\{x \mathrel{\hat{=}} 4\})]^n \cdot [(x \mathrel{\hat{=}} 4, \varnothing) \rightarrowtail x \mathrel{\hat{=}} 4]^2 \mid n \in \mathbf{N}\} \sqcup \\ & \{[stutt(\{x \mathrel{\hat{=}} 4\})]^{+\infty}\} \end{cases}$$

$$\mathcal{D}^{\pm}[\![D]\!]{\uparrow}2 = \begin{cases} p(x) \mapsto & \{[stutt(\{x \mathrel{\hat{=}} 4\})]^n \cdot [(x \mathrel{\hat{=}} 4, \varnothing) \rightarrowtail x \mathrel{\hat{=}} 4]^4 \mid n \in \mathbf{N}\} \sqcup \\ & \{[stutt(\{x \mathrel{\hat{=}} 4\})]^{+\infty}\} \end{cases}$$

$$\vdots$$

$$\mathcal{F}^{\pm}[\![D]\!] = \begin{cases} p(x) \mapsto & \bigvee\big\{[stutt(\{x \mathrel{\hat{=}} 4\})]^n \cdot [(x \mathrel{\hat{=}} 4, \varnothing) \rightarrowtail x \mathrel{\hat{=}} 4]^{+\infty} \mid n \geq 0\}\big\} \vee \\ & \{[stutt(\{x \mathrel{\hat{=}} 4\})]^{+\infty}\} \end{cases}$$

Comparing this result to the abstraction $\alpha(\mathcal{F}[\![D]\!])$ of Example 4.2.12, it is worth noticing that $\alpha(\mathcal{F}[\![D]\!]) = \mathcal{F}^{\pm}[\![D]\!]$, i.e., the abstract fixpoint coincides with the abstraction of the concrete fixpoint.

[2]Correct means $\tilde{\alpha}(\mathcal{F}^{\pm}[\![D]\!]) \leq \mathcal{F}[\![D]\!]$ and best means that is the minimum (w.r.t. $\leq$) of all correct approximations.

Example 4.3.11

Consider Constraint System 1.4.3 and declaration D of Example 3.1.21. The first iteration of the operator $\mathcal{D}$ gives rise to the following set:

$$\mathcal{D}^{\pm}[\![D]\!]\!\uparrow\! 1 = \begin{cases} \{q(x,y) \mapsto \{[(\mathrm{pos}_x,\varnothing) \rightarrowtail \mathrm{pos}_x \mathbin{\hat{\otimes}} \mathrm{neg}_y]^1 \cdot \boxtimes, \\ \qquad\qquad\quad [(\widehat{true},\{\mathrm{pos}_x\}) \rightarrowtail \widehat{true}]^1\} \end{cases}$$

$$\mathcal{D}^{\pm}[\![D]\!]\!\uparrow\! 2 = \begin{cases} q(x,y) \mapsto \{[(\mathrm{pos}_x,\varnothing) \rightarrowtail \mathrm{pos}_x \mathbin{\hat{\otimes}} \mathrm{neg}_y]^1 \cdot \boxtimes\} \vee \\ \qquad\qquad \{[(\widehat{true},\{\mathrm{pos}_x\}) \rightarrowtail \widehat{true}]^1 \cdot [(\mathrm{pos}_x,\varnothing) \rightarrowtail \mathrm{pos}_x \mathbin{\hat{\otimes}} \mathrm{neg}_y]^1 \cdot \boxtimes\} \vee \\ \qquad\qquad \{[(\widehat{true},\{\mathrm{pos}_x\}) \rightarrowtail \widehat{true}]^2\} \end{cases}$$

$$\vdots$$

$$\mathcal{D}^{\pm}[\![D]\!]\!\uparrow\! n = \begin{cases} \{q(x,y) \mapsto \{[(\mathrm{pos}_x,\varnothing) \rightarrowtail \mathrm{pos}_x \mathbin{\hat{\otimes}} \mathrm{neg}_y]^1 \cdot \boxtimes\} \vee \\ \qquad\qquad \{[(\widehat{true},\{\mathrm{pos}_x\}) \rightarrowtail \widehat{true}]^{n-1} \cdot [(\mathrm{pos}_x,\varnothing) \rightarrowtail \mathrm{pos}_x \mathbin{\hat{\otimes}} \mathrm{neg}_y]^1 \cdot \boxtimes\} \vee \\ \qquad\qquad \{[(\widehat{true},\{\mathrm{pos}_x\}) \rightarrowtail \widehat{true}]^n\} \end{cases}$$

Finally, the limit of the computation is the following set:

$$\mathcal{F}^{\pm}[\![D]\!] = \begin{cases} \{q(x,y) \mapsto \bigvee\{[(\widehat{true},\{\mathrm{pos}_x\}) \rightarrowtail \widehat{true}]^n \cdot [(\mathrm{pos}_x,\varnothing) \rightarrowtail \mathrm{pos}_x \mathbin{\hat{\otimes}} \mathrm{neg}_y]^1 \cdot \boxtimes \mid n \geq 0\} \\ \qquad\qquad \vee \{[(\widehat{true},\{\mathrm{pos}_x\}) \rightarrowtail \widehat{true}]^{+\infty}\} \end{cases}$$

Intuitively, the abstract behavior says that the program waits until x is positive. Then, as soon as this information is (abstractly) entailed from the store, then the constraint neg_y is added.

Notice that the guard in the (concrete) program is $x > 2$, so the abstract semantics is less restricted than the concrete one.

Finally, if we compare this set with the abstraction $\tilde{\alpha}(\mathcal{F}[\![D]\!])$ of Example 4.2.13, we can see that $\tilde{\alpha}(\mathcal{F}[\![D]\!]) = \mathcal{F}^{\pm}[\![D]\!]$, i.e., the abstract fixpoint coincides with the abstraction of the concrete fixpoint.

4.4 Abstract Diagnosis for *tccp* based on constraint system abstractions

In this section we present some examples of the abstract diagnosis approach for *tccp* introduced in Section 4.1. We use the abstraction framework of Section 4.2 and the corresponding induced abstract semantics defined in Section 4.3. Therefore, specifications are given in terms of sets of compact abstract conditional traces. Notice that, in this way, it is possible to specify infinite behaviors just by using the index $+\infty$ in a compact abstract conditional state.

Let us recall that, although the domain of compact conditional abstract traces is not noetherian, starting from a finite interpretation, a single step of the immediate consequence operator $\mathcal{D}^{\pm}[\![D]\!]$ is computed in finite time. For this reason, the proposed abstract diagnosis check for this domain is decidable if the given specification is finite.

The first example shows how we can deal with the constructors that introduce the non-monotonic behavior of the system, in particular the now agent. This is a novel contribution since, to our knowledge, the previous diagnosis proposals for the timed extensions of the *cc* paradigm cannot address this difficulty.

Example 4.4.1

We model a (simplified) time-out(n) process that checks during at most n times units if the system emits a signal telling that the process evolves normally ($system = ok$). When the signal arrives, the system emits the fact that there is no alert ($alert = no$)[3]. Consider the following declarations:

$$d_0 ::= \mathit{time\text{-}out}(0) :- \mathsf{now}\ system = ok\ \mathsf{then}\ action\ \mathsf{else}\ (\mathsf{ask}(true) \to \mathit{time\text{-}out}(0))$$
$$d_n ::= \mathit{time\text{-}out}(n) :- \mathsf{now}\ system = ok\ \mathsf{then}\ action\ \mathsf{else}\ (\mathsf{ask}(true) \to \mathit{time\text{-}out}(n-1))$$
$$d_{action} ::= action :- \mathsf{tell}(alert = no)$$

When the time limit is reached (declaration d_0), the system should set the signal *alert* to *yes* ($\mathsf{tell}(alert = no)$). However, we have introduced an error in the program, by recursively invoking the process $\mathit{time\text{-}out}(0)$ instead.

Due to the simplicity of the constraint system, the abstract domain coincides with the concrete one, and the two external functions are the $\hat{\oplus}$ and $\check{\oplus}$ operators.

Let us now consider the following specification. For d_0 we expect that, if the *ok* signal is present, then it ends with an $alert = no$ signal, otherwise an alert should be emitted. This is represented by two possible sequences, one with a condition where $system = ok$, and a second one when $system = ok$ is absent (i.e., a sequence that reasons with the *absence* of information).

$$\begin{aligned}\mathcal{S}^{\pm}(\mathit{time\text{-}out}(0)) = \{ & [(system = ok, \widecheck{false}) \rightarrowtail system = ok]^1 \cdot \\ & [(\widehat{true}, \widecheck{false})) \rightarrowtail system = ok \mathbin{\hat{\otimes}} alert = no]^1 \cdot \boxtimes \} \\ \cup \{ & [(\widehat{true}, \{system = ok\}) \rightarrowtail \widehat{true}]^1 \cdot [(\widehat{true}, \widecheck{false}) \rightarrowtail alert = yes]^1 \cdot \boxtimes \}\end{aligned}$$

The specification for d_n is similar, but we add n sequences, since we have the possibility that the signal arrives at each time instant before n.

$$\begin{aligned}\mathcal{S}^{\pm}(\mathit{time\text{-}out}(n)) = \{ & [(\widehat{true}, \{system = ok\}) \rightarrowtail \widehat{true}]^m \cdot \\ & [(system = ok, \widecheck{false}) \rightarrowtail system = ok]^1 \cdot \\ & [(\widehat{true}, \widecheck{false}) \rightarrowtail system = ok \mathbin{\hat{\otimes}} alert = no]^1 \cdot \boxtimes \mid 0 \le m < n \} \\ \cup \ \{ & [(\widehat{true}, \{system = ok\}) \rightarrowtail \widehat{true}]^{n+1} \cdot [(\widehat{true}, \widecheck{false}) \rightarrowtail alert = yes]^1 \cdot \boxtimes \} \\ \mathcal{S}^{\pm}(action) = \{ & [(\widehat{true}, \widecheck{false}) \rightarrowtail alert = no]^1 \cdot \boxtimes \}\end{aligned}$$

Now, when we compute $\mathcal{D}^{\pm}[\![\{d_0\}]\!]_{\mathcal{S}^{\pm}}$ we have:

$$\begin{aligned}& \{[(system = ok, \widecheck{false}) \rightarrowtail system = ok]^1 \cdot [(\widehat{true}, \widecheck{false}) \rightarrowtail system = ok \mathbin{\hat{\otimes}} alert = no]^1 \cdot \boxtimes \} \\ & \cup \ \{[(\widehat{true}, \{system = ok\}) \rightarrowtail \widehat{true}]^1 \cdot [(system = ok, \widecheck{false}) \rightarrowtail system = ok]^1 \cdot \\ & \quad [(\widehat{true}, \widecheck{false}) \rightarrowtail system = ok \mathbin{\hat{\otimes}} alert = no]^1 \cdot \boxtimes \} \\ & \cup \ \{[(\widehat{true}, \{system = ok\}) \rightarrowtail \widehat{true}]^2 \cdot [(\widehat{true}, \widecheck{false}) \rightarrowtail alert = no]^1 \cdot \boxtimes \}\end{aligned}$$

Due to the last sequence, $\mathcal{D}^{\pm}[\![\{d_0\}]\!]_{\mathcal{S}^{\pm}} \not\le \mathcal{S}^{\pm}$, so we conclude that d_0 is (abstractly) incorrect. This error is provoked by the recursive call in the else branch of d_0. If we fix the program by replacing d_0 by d_0' where the recursive call is replaced by $\mathsf{tell}(alert = yes)$, then $\mathcal{D}^{\pm}[\![\{d_0'\}]\!]_{\mathcal{S}^{\pm}} \le \mathcal{S}^{\pm}$, thus d_0' is abstractly correct.

[3]The classical *time-out* would restart the countdown by recursively calling time-out(n).

The second example illustrates how one can work with the abstraction of the constraint system, and also how we can take advantage of our abstract domain.

Example 4.4.2

Let us consider a system with a single declaration and the abstraction of the constraint system that abstracts integer variables to a (simplified) interval-based domain with abstract values $\{\top, \mathrm{pos}_x, \mathrm{neg}_x, x{>}10, x{\leq}10, \bot\}$.

$$\begin{aligned} d ::= p(x) :\!- \ &\mathsf{now}\ (x \dot{>} 0)\ \mathsf{then}\ \exists x'\,(\mathsf{tell}(x = [\_\,|\,x']) \parallel \mathsf{tell}(x' = [x+1\,|\,\_]) \parallel p(x')) \\ &\mathsf{else}\ \exists x''\,(\mathsf{tell}(x = [\_\,|\,x'']) \parallel \mathsf{tell}(x'' = [x-1\,|\,\_]) \parallel p(x'')) \end{aligned}$$

Again, we have to use streams to model the *imperative-style* variables [43]. In this way, variable x in the program above is a stream that is updated with different values during the execution. Following this idea, the abstraction for concrete streams is defined as the (abstracted) last instantiated value in the stream. The concretization of one stream is defined as all the concrete streams whose last value is a concretization of the abstract one. We write a dot on a predicate symbol (e.g. $\dot{=}$) to denote that we want to check it for the last instantiated value of a stream.

We define the following intended specification to specify that, (a) if the value of x in the initial call is greater than 10, then the last value of the stream (written $\dot{x}$) will always be greater than 10; (b) if the value is negative, then the value is always negative

$$\mathcal{S}^{\pm}(p(x_1) = \{[(x_1 \dot{>} 10, \check{false}) \rightarrowtail x \dot{>} 10]^{+\infty}\} \cup \{[(\mathrm{neg}_{\dot{x}}, \check{false}) \rightarrowtail \mathrm{neg}_{\dot{x}}]^{+\infty}\}$$

The two abstract sequences represent infinite computations thanks to the $+\infty$ index in the last tuple. In other words, finite specifications that represent infinite computations can be considered and effectively handled. In fact, we can compute $\mathcal{D}^{\pm}[\![\{d\}]\!]_{\mathcal{S}^{\pm}}$:

$$\begin{aligned} &\{\,\{[(\mathrm{neg}_{\dot{x}}, \check{false}) \rightarrowtail \mathrm{pos}_{\dot{x}}]^1 \cdot ([(\dot{x}{>}10, \check{false}) \rightarrowtail \dot{x}{>}10]^{+\infty} \tilde{\downarrow}_{\mathrm{pos}_{\dot{x}}})\} \\ &\quad \cup \{[(\mathrm{neg}_{\dot{x}}, \check{false}) \rightarrowtail \mathrm{neg}_{\dot{x}}]^1 \cdot [(\mathrm{neg}_{\dot{x}}, \check{false}) \rightarrowtail \mathrm{neg}_{\dot{x}}]^{+\infty}\}\} \\ &= \\ &\{\,\{[(\mathrm{neg}_{\dot{x}}, \check{false}) \rightarrowtail \mathrm{pos}_{\dot{x}}]^1 \cdot [(\overbrace{\mathrm{pos}_{\dot{x}} \mathbin{\hat{\otimes}} \dot{x}{>}10}^{\mathrm{pos}_{\dot{x}}}, \check{false}) \rightarrowtail \mathrm{pos}_{\dot{x}} \mathbin{\hat{\otimes}} \dot{x}{>}10]^{+\infty})\} \\ &\quad \cup \{[(\mathrm{neg}_{\dot{x}}, \check{false}) \rightarrowtail \mathrm{neg}_{\dot{x}}]^1 \mathrm{neg}_{\dot{x}} \cdot [(\mathrm{neg}_{\dot{x}}, \check{false}) \rightarrowtail \mathrm{neg}_{\dot{x}}]^{+\infty} \mathrm{neg}_{\dot{x}}\}\} \\ &= \\ &\{\,\{[(\mathrm{pos}_{\dot{x}}, \check{false}) \rightarrowtail \mathrm{pos}_{\dot{x}}]^{+\infty}\} \cup \{[(\mathrm{neg}_{\dot{x}}, \check{false}) \rightarrowtail \mathrm{neg}_{\dot{x}}]^{+\infty} \mathrm{neg}_{\dot{x}}\}\} \end{aligned}$$

The third equality holds because $\mathrm{pos}_{\dot{x}}$ entails $x \dot{>} 10$, so the merge of the two constraints will be equal to $\mathrm{pos}_{\dot{x}}$.

Since $\mathcal{D}^{\pm}[\![\{d\}]\!]_{\mathcal{S}^{\pm}} \not\leq \mathcal{S}^{\pm}$ we can conclude that d is an incorrect declaration w.r.t. $\mathcal{S}^{\pm}$. In addition, we can notice that $\mathcal{S}^{\pm}$ contains an uncovered element that is a sequence that cannot be derived by the semantics operator $\mathcal{D}^{\pm}$.

Our third example shows a system already studied in [51]. The most important difference w.r.t. the time-out example is that the *control* process of this example needs that someone explicitly tells the system that an error has occurred by telling *failure*. Instead, in the time-out example, the system is able to act (and maybe recover) when it detects that something that should have happened, had not. In other words, the *control* example does not handle *absence* of information, since non-monotonic operators are not considered there. We have implemented the example in *tccp* and we have checked that the same results as in [51] can be achieved in our framework if we apply the same abstraction they use: a $depth(k)$ abstraction. This abstraction cuts the traces at a given depth k, this corresponds to check for the interested property just up to a given time instant k. It can be noticed that, by using this abstraction, the ability of checking infinite behaviors is lost.

Example 4.4.3

Let D be a set containing the following declarations (d_1 and d_2, respectively). The idea of the system is to control, at each time instant, if a failure signal has arrived. In that case, an action is taken (for instance just a constraint *stop* is added).

$$\begin{aligned} d_1 ::= \mathit{control}(i,o) \mathrel{:-} \exists o', i' \;\; & \mathsf{now}\ i = [\mathit{fail}|\_]\ \mathsf{then}\ (\mathsf{tell}(i = [\mathit{fail}|i']) \parallel \mathit{action}(o,o')) \\ & \qquad\qquad\qquad \mathsf{else\ skip} \\ & \parallel\ \mathsf{ask}(\mathit{true}) \to \mathit{control}(i',o') \\ d_2 ::= \mathit{action}(o,o') \mathrel{:-} & \mathsf{tell}(o = [\mathit{stop}|o']) \end{aligned}$$

The concrete domain for the constraint system is composed by the elements *fail*, *stop*, *true* and *false*. The abstract setting is similar to the one in the previous example. Due to the monotonicity of the store, we have to use streams to model the *imperative-style* variables [43]. As in the previous example, we write a dot on a predicate symbol (e.g. $\doteq$) to denote that we want to check it for the last instantiated value of a stream. The abstraction for concrete streams is defined as the last instantiated value in the stream (see Example 4.2.3).

Let us now check that the *action* process finishes in one time instant. To this end, we define the following specification: $\mathcal{S}^{\pm}(\mathit{action}(x_1,x_2)) = \{[(\hat{\mathit{true}}, \check{\mathit{false}}) \rightarrowtail x_1 \doteq \mathit{stop}]^1 \cdot \boxtimes\}$. If we compute one iteration of the semantic operator $\mathcal{D}^{\pm}$ on the specification, we get $\mathcal{D}^{\pm}[\![\{d_2\}]\!]_{\mathcal{S}^{\pm}} = \{[(\hat{\mathit{true}}, \check{\mathit{false}}) \rightarrowtail \tau^{+}(o = [\mathit{stop}|o'])]^1 \cdot \boxtimes\} = \{[(\hat{\mathit{true}}, \check{\mathit{false}}) \rightarrowtail o \doteq \mathit{stop}]^1 \cdot \boxtimes\}$, thus we conclude that the declaration d_2 is correct w.r.t $\mathcal{S}^{\pm}$.

Let us now define the specification for the *control* process:

$$\begin{aligned} \mathcal{S}^{\pm}(\mathit{control}(f,s)) = \{ & [(\hat{\mathit{true}}, \{f \doteq \mathit{fail}\}) \rightarrowtail \hat{\mathit{true}}]^n \cdot \\ & [(f \doteq \mathit{fail}, \check{\mathit{false}}) \rightarrowtail f \doteq \mathit{fail} \mathbin{\hat{\otimes}} \hat{\exists}_{f'}(f' \doteq \mathit{true})]^1 \cdot \\ & [(\hat{\mathit{true}}, \check{\mathit{false}}) \rightarrowtail f \doteq \mathit{fail} \mathbin{\hat{\otimes}} \hat{\exists}_{f'}(f' \doteq \mathit{true} \mathbin{\hat{\otimes}} s \doteq \mathit{stop})]^1 \cdot \boxtimes \mid n \in \mathbf{N}\} \\ \cup \{ & [(\hat{\mathit{true}}, \{f \doteq \mathit{fail}\}) \rightarrowtail \hat{\mathit{true}}]^{+\infty}\} \end{aligned}$$

Note that this specification is infinite since the first set of traces ranges over natural numbers. In order to make the abstract diagnosis process effective, one solution would be to use our framework with a (much concrete) $depth(k)$abstraction (similar to what is done in [51]) in order to check only up to a given time instant k. We remark that this is not equivalent to model-check (for instance) an equivalent temporal property (written in some temporal logic). We will overcome this problem in Chapter 5 where we define an abstract diagnosis method for *tccp* by using temporal formulas as specifications. In that framework, the above specification for $control(f,s)$ is simply specified by the formula $\square(f \doteq \mathit{fail} \mathbin{\dot{\to}} \bigcirc^2 s \doteq \mathit{stop})$ which says that always in the future if the last instantiated value of f is *fail* then, after two instants of time, the last value of s becomes *stop*.

4.5 Related Work

In [51], a first approach to the declarative debugging of a *ccp* language is presented. The authors introduce a denotational semantics to reason about *ntcc* programs, and, in order to make their debugging approach effective, they approximate the behavior of the program by using an abstract domain which cut the infinite behavioral sequences at a given depth. In [50], the same method is proposed for the *utcc* language.

This approach has some drawbacks. First of all, it does not cover the particular extra difficulty of modeling the semantics of non-monotonic operators, common to all timed concurrent constraint languages. As already pointed out, this ability is crucial in order to model specific behaviors of reactive systems, such as timeouts or preemption actions (see Example 4.4.1).

Second, infinite sequences are approximated by cutting them at a given depth. Thus, it is not possible to verify with enough precision infinite behaviors which, in our opinion, are essential in the context of reactive systems.

Furthermore, the concrete and abstract semantics used in [51, 50] are not condensed. This might cause some practical problems if the considered underlying constraint system is infinite. In fact, the immediate consequence operator can possibly generate an infinite number of (finite) behavioral sequences making the diagnosis check not decidable.

These are the main reasons why our abstract diagnosis approach is significantly different from the one presented in [51, 50].

The idea of using two different mechanisms for dealing with positive and negative information in our abstraction scheme is inspired by [4]. There, a framework for the abstract model checking of *tccp* programs based on a source-to-source transformation is defined. In particular, it is defined a transformation from a *tccp* program P into a *tccp* program $\bar{P}$ that represents a correct abstraction of the original one (in the sense that the semantics of P are included in the semantics of $\bar{P}$). Instead, we define an abstract semantics for the language. The upper- and lower-approximated versions of the entailment relation are used in order to keep $\bar{P}$ correct, but also precise enough.

4.6 Discussion on the results

In this chapter, we have first formally introduced a generic framework for the abstract diagnosis of *tccp* programs, which is parametric w.r.t. the chosen abstract domain. Then, we have instantiated this framework with a suitable abstract domain and the correspondent abstract semantics which models the full *tccp* language and that is able to deal with infinite computations.

Among other valuable facilities, abstract diagnosis supports the development of efficacious diagnostic tools that detect program errors automatically without having to determine symptoms in advance.

Because of the compositional nature of the underlying concrete semantics, our proposal can be used with *partial specifications* and also with *partial programs.* Obviously, one cannot detect errors in process declarations involving processes which have not been specified, but for the process declarations that involve processes that have a specification, the check can be made, even if the whole program has not been written yet. With other "global" approaches such programs could not be checked at all. This is particularly useful for applications, since the diagnosis could be used from the beginning of the development phase. Moreover, it could be performed incrementally, thus the overall computational cost can be parceled over time.

Nevertheless, the main drawback of abstract diagnosis (and in general of all approximation based methods) is the loss of precision due to the semantics abstraction. In fact, because of the approximation, it can happen that a (concretely) correct program is marked as (abstractly) incorrect, generating a false positive. However, all concrete errors are assured to be detected.

By instantiating the general abstract diagnosis framework with the abstract semantics of Section 4.3 we obtain a new debugging method for *tccp*. Our method keeps the information about infinite behavioral traces and it is able to deal with full *tccp* including non-monotonic operators. In fact, differently from the approach presented in [51, 50], we do not make any restriction on the program syntax. As we have said, these abilities are crucial in order to model and verify interesting properties of reactive systems.

4.A Proofs

4.A.1 Proofs of Section 4.1

Theorem 4.1.3. *Let $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$ and $\mathcal{S}^\alpha \in \mathbf{I_A}$.*

1. *If there are no abstractly incorrect process declarations in D (i.e., $\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$), then D is partially correct w.r.t. $\mathcal{S}^\alpha$ (i.e., $\alpha(\mathcal{F}[\![D]\!]) \leq \mathcal{S}^\alpha$).*

2. *Let D be partially correct w.r.t. $\mathcal{S}^\alpha$. If D has abstract uncovered elements, then D is not complete (i.e., $\mathcal{S}^\alpha \nleq \alpha(\mathcal{F}[\![D]\!])$).*

Proof.

Point 1 By hypothesis, $\forall r \in D.\, \mathcal{D}^\alpha[\![\{r\}]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$. Hence $\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$, i.e., $\mathcal{S}^\alpha$ is a pre-fixpoint of $\mathcal{D}^\alpha[\![D]\!]$. Since $\alpha(\mathcal{F}[\![D]\!]) \leq \mathcal{F}^\alpha[\![D]\!] = \mathit{lfp}\,\mathcal{D}^\alpha[\![D]\!]$, by Knaster–Tarski's Theorem $\alpha(\mathcal{F}[\![D]\!]) \leq \mathcal{F}^\alpha[\![D]\!] \leq \mathcal{S}^\alpha$. The thesis follows by definition of correctness.

Point 2 By construction, $\alpha \circ \mathcal{D}[\![D]\!] \circ \gamma \leq \mathcal{D}^\alpha[\![D]\!]$, hence $\alpha \circ \mathcal{D}[\![D]\!] \circ \gamma \circ \alpha \leq \mathcal{D}^\alpha[\![D]\!] \circ \alpha$. Since $\mathit{id} \sqsubseteq \gamma \circ \alpha$, it holds that $\alpha \circ \mathcal{D}[\![D]\!] \leq \alpha \circ \mathcal{D}[\![D]\!] \circ \gamma \circ \alpha$ and $\alpha \circ \mathcal{D}[\![D]\!] \leq \mathcal{D}^\alpha[\![D]\!] \circ \alpha$. Hence,

$$\begin{aligned}
&\alpha(\mathcal{F}[\![D]\!]) = && [\text{ since } \mathcal{F}[\![D]\!] \text{ is a fixpoint }]\\
&\alpha(\mathcal{D}[\![D]\!]_{\mathcal{F}[\![D]\!]}) \leq && [\text{ by } \alpha \circ \mathcal{D}[\![D]\!] \leq \mathcal{D}^\alpha[\![D]\!] \circ \alpha\,]\\
&\mathcal{D}^\alpha[\![D]\!]_{\alpha(\mathcal{F}[\![D]\!])} \leq && [\text{ since } \mathcal{D}^\alpha[\![D]\!] \text{ is monotone and } D \text{ is partial correct }]\\
&\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha}
\end{aligned}$$

Now, if D has an abstract uncovered element e i.e., $e \leq \mathcal{S}^\alpha$ and $e \wedge \mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} = \bot$, then $e \wedge \alpha(\mathcal{F}[\![D]\!]) = \bot$ and $\mathcal{S}^\alpha \nleq \alpha(\mathcal{F}[\![D]\!])$. The thesis follows from definition of completeness.

Theorem 4.1.4. *Let r be a process declaration and $\mathcal{S}$ a concrete specification.*

1. *If $\mathcal{D}[\![\{r\}]\!]_{\mathcal{S}} \nleq \mathcal{S}$ and $\alpha(\mathcal{D}[\![\{r\}]\!]_{\mathcal{S}}) \nleq \alpha(\mathcal{S})$, then r is abstractly incorrect w.r.t. $\alpha(\mathcal{S})$.*

2. *If there exists an abstract uncovered element a w.r.t. $\alpha(\mathcal{S})$, such that $\gamma(a) \sqsubseteq \mathcal{S}$ and $\gamma(\bot) = \{\epsilon\}$, then there exists a concrete uncovered element c w.r.t. $\mathcal{S}$ (i.e., $c \sqsubseteq \mathcal{S}$ and $c \sqcap \mathcal{D}[\![D]\!]_{\mathcal{S}} = \{\epsilon\}$).*

Proof.

Point 1 Since $\mathcal{S} \sqsubseteq \gamma \circ \alpha(\mathcal{S})$, by monotonicity of α and the correctness of $\mathcal{D}^{\alpha}[\![\{r\}]\!]$, it holds that $\alpha(\mathcal{D}[\![\{r\}]\!]_{\mathcal{S}}) \leq \alpha(\mathcal{D}[\![\{r\}]\!]_{\gamma \circ \alpha(\mathcal{S})}) \leq \mathcal{D}^{\alpha}[\![\{r\}]\!]_{\alpha(\mathcal{S})}$. By hypothesis $\alpha(\mathcal{D}[\![\{r\}]\!]_{\mathcal{S}}) \nleq \alpha(\mathcal{S})$, therefore $\mathcal{D}^{\alpha}[\![\{r\}]\!]_{\alpha(\mathcal{S})} \nleq \alpha(\mathcal{S})$ since $\alpha(\mathcal{D}[\![\{r\}]\!]_{\mathcal{S}}) \leq \mathcal{D}^{\alpha}[\![\{r\}]\!]_{\alpha(\mathcal{S})}$. The thesis holds by Definition 4.1.2.

Point 2 By hypothesis, $a \leq \alpha(\mathcal{S})$ and $a \wedge \mathcal{D}^{\alpha}[\![D]\!]_{\alpha(\mathcal{S})} = \bot$. Hence $\gamma(a) \sqcap \gamma(\mathcal{D}^{\alpha}[\![D]\!]_{\alpha(\mathcal{S})}) = \{\epsilon\}$ since $\gamma(\bot) = \{\epsilon\}$ and γ preserves greatest lower bounds. By construction $\mathcal{D}^{\alpha}[\![D]\!] = \alpha \circ \mathcal{D}[\![D]\!] \circ \gamma$, thus $\gamma(a) \sqcap \gamma(\alpha(\mathcal{D}[\![D]\!]_{\gamma(\alpha(\mathcal{S}))})) = \{\epsilon\}$. Since $id \sqsubseteq \gamma \circ \alpha$ and by monotonicity of $\mathcal{D}[\![D]\!]$, $\gamma(a) \sqcap \mathcal{D}[\![D]\!]_{\mathcal{S}} = \{\epsilon\}$. By hypothesis, $\gamma(a) \sqsubseteq \mathcal{S}$ hence $\gamma(a)$ is a concrete uncovered element.

4.A.2 Proofs of Section 4.2

Lemma 4.2.8. *$(\alpha^{\pm}, \gamma^{\pm})$ is a Galois Insertion*

$$(\mathbf{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}, \{\epsilon\}) \underset{\alpha^{\pm}}{\overset{\gamma^{\pm}}{\leftrightarrows}} (\mathbf{M}^{\pm}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}^{\pm}, \{\epsilon\})$$

Proof.

$\alpha^{\pm}$ is monotonic Let be $R, R' \in \mathbf{M}$ such that $R \sqsubseteq R'$. We have that

$$\alpha^{\pm}(R) = \bigsqcup\{\alpha^{\pm}(r) \mid r \in R\} \sqsubseteq \bigsqcup\{\alpha^{\pm}(r) \mid r \in R'\} = \alpha^{\pm}(R').$$

$\gamma^{\pm}$ is monotonic Consider $\tilde{R}, \tilde{R}' \in \mathbf{M}^{\pm}$ such that $\tilde{R} \sqsubseteq \tilde{R}'$. It follows that

$$\gamma^{\pm}(\tilde{R}) = \bigsqcup\{\gamma^{\pm}(\tilde{r}) \mid \tilde{r} \in \tilde{R}\} \sqsubseteq \bigsqcup\{\gamma^{\pm}(\tilde{r}) \mid \tilde{r} \in \tilde{R}'\} = \gamma^{\pm}(\tilde{R}').$$

$(\gamma^{\pm} \circ \alpha^{\pm})$ is extensive This means that $\forall R \in \mathbf{M}.\ R \sqsubseteq \gamma^{\pm}(\alpha^{\pm}(R))$. Thus, we show that for all $r \in R$ it exists $\bar{r} \in \gamma^{\pm}(\alpha^{\pm}(R))$ such that r is a prefix of $\bar{r}$. We proceed by structural induction on r.

$r = \epsilon$ From Definition 4.2.7, it follows directly that $\epsilon \in \alpha^{\pm}(R)$ and $\epsilon \in \gamma^{\pm}(\alpha^{\pm}(R))$.

$r = \boxtimes$ By Definition 4.2.7, we have that $\boxtimes \in \alpha^{\pm}(R)$ and, thus, $\boxtimes \in \gamma^{\pm}(\alpha^{\pm}(R))$.

$r = (\eta^{+}, \eta^{-}) \rightarrowtail c \cdot r'$ By (4.2.5) it follows that $\alpha^{\pm}(r) = (\tau^{+}(\eta^{+}), \tau^{-}(\eta^{-})) \rightarrowtail \tau^{+}(c) \cdot \alpha^{\pm}(r')$. For the properties expressed by (4.2.2) the condition $(\tau^{+}(\eta^{+}), \tau^{-}(\eta^{-}))$ is valid, thus also $(\tau^{+}(\eta^{+}), \tau^{-}(\eta^{-})) \rightarrowtail \tau^{+}(c)$ is valid. Furthermore, by (4.2.1e) and (4.2.1i), there follow the properties of monotonicity and consistency for $\alpha^{\pm}(r)$, thus it belongs to the domain $\mathbf{M}^{\pm}$. By (4.2.9), it follows that $r \in \gamma^{\pm}(\alpha^{\pm}(R))$.

$r = \mathit{stutt}(\eta^{-}) \cdot r'$ By (4.2.5) it follows that $\alpha^{\pm}(r) = \mathit{stutt}(\tau^{-}(\eta^{-})) \cdot \alpha^{\pm}(r')$. As in the previous case from (4.2.1e) and (4.2.1i) we can say that this sequence is a valid one and we can conclude that $r \in \gamma^{\pm}(\alpha^{\pm}(R))$.

$(\alpha^{\pm} \circ \gamma^{\pm})$ is the identity for $\mathbf{M}^{\pm}$ We show that $\forall \tilde{R} \in \mathbf{M}^{\pm}.\ \tilde{R} = (\alpha^{\pm} \circ \gamma^{\pm})(\tilde{R})$.

$\subseteq$ We first prove that for all $\tilde{r} \in \tilde{R}$, $\tilde{r} \in \alpha^{\pm}(\gamma^{\pm}(\tilde{R}))$ by structural induction on $\tilde{r}$.

$\tilde{r} = \epsilon$ By (4.2.9), $\epsilon \in \gamma^{\pm}(\tilde{R})$ and by (4.2.5), $\epsilon \in \alpha^{\pm}(\gamma^{\pm}(\tilde{R}))$.

$\tilde{r} = \boxtimes$ By (4.2.9), $\boxtimes \in \gamma^{\pm}(\tilde{R})$ and by (4.2.5), $\boxtimes \in \alpha^{\pm}(\gamma^{\pm}(\tilde{R}))$.

$\tilde{r} = \tilde{\eta} \rightarrowtail \hat{c} \cdot \tilde{r}'$ From (4.2.9) it follows that $(\eta^{+}, \eta^{-}) \rightarrowtail c \cdot r' \in \gamma^{\pm}(\tilde{R})$ where $\tau^{+}(\eta^{+}) = \hat{\eta}$, $\tau^{-}(\eta^{-}) = \check{\eta}$, $\tau^{+}(c) = \hat{c}$ and $r' \in \gamma^{\pm}(\tilde{r}')$. Thus, by (4.2.5) and by inductive hypothesis, it follows that $\tilde{r} \in \alpha^{\pm}(\gamma^{\pm}(\tilde{R}))$.

$\tilde{r} = stutt(\check{\eta}) \cdot \tilde{r}'$ From (4.2.9) it follows that $stutt(\eta^{-}) \cdot r' \in \gamma^{\pm}(\tilde{R})$ where $\tau^{-}(\eta^{-}) = \check{a}$ and $r' \in \gamma^{\pm}(\tilde{r}')$. Therefore, by (4.2.5) and by inductive hypothesis, we have that $\tilde{r} \in \alpha^{\pm}(\gamma^{\pm}(\tilde{R}))$.

$\supseteq$ Now we show the other inclusion, for all $\tilde{r} \in \alpha^{\pm}(\gamma^{\pm}(\tilde{R}))$, $\tilde{r} \in \tilde{R}$. We proceed by structural induction on $\tilde{r}$.

$\tilde{r} = \epsilon$ By (4.2.5), $\epsilon \in \gamma^{\pm}(\tilde{R})$ and from (4.2.9) it follows directly that $\epsilon \in \tilde{R}$.

$\tilde{r} = \boxtimes$ By (4.2.5), $\boxtimes \in \gamma^{\pm}(\tilde{R})$ and from (4.2.9) it follows directly that $\boxtimes \in \tilde{R}$.

$\tilde{r} = \tilde{\eta} \rightarrowtail \hat{c} \cdot \tilde{r}'$ By (4.2.5), it can be noticed that it exists a concrete sequence $r \in \gamma^{\pm}(\tilde{R})$ such that $r = (\eta^{+}, \eta^{-}) \rightarrowtail c \cdot r'$ and $\tau^{+}(\eta^{+}) = \hat{\eta}$, $\tau^{-}(\eta^{-}) = \check{\eta}$, $\tau^{+}(c) = \hat{c}$ and $\alpha^{\pm}(r') = \tilde{r}'$. But this means that $\tilde{\eta} \rightarrowtail \hat{a} \cdot \tilde{r}' \in \tilde{R}$, otherwise we would not obtain r by applying $\gamma^{\pm}$ to $\tilde{R}$.

$\tilde{r} = stutt(\check{\eta}) \cdot \tilde{r}'$ By (4.2.5), it can be noticed that it exists a concrete sequence $r \in \gamma^{\pm}(\tilde{R})$ such that $r = stutt(\eta^{-}) \cdot r'$, $\tau^{-}(\eta^{-}) = \check{\eta}$ and $\alpha^{\pm}(r') = \tilde{r}'$. This means that $stutt(\check{\eta}) \cdot \tilde{r}' \in \tilde{R}$ otherwise r would not be obtained by applying $\gamma^{\pm}$ to $\tilde{R}$.

Lemma 4.2.11. *(κ, κ^{-1}) is an order-preserving isomorphism*

$$(\mathbf{M}^{\pm}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{M}^{\pm}, \{\epsilon\}) \underset{\kappa}{\overset{\kappa^{-1}}{\leftrightarrows}} (\mathbf{A}, \leq, \bigvee, \bigwedge, \top, \bot).$$

Proof.

$(\kappa^{-1} \circ \kappa)$ is the identity for $\mathbf{M}^{\pm}$. Let $\bar{R} \in \mathbf{M}^{\pm}$, then

$$\begin{aligned}
\kappa^{-1}(\kappa(\bar{R})) &= \kappa^{-1}(\bigvee\{\kappa(\bar{r}) \mid \bar{r} \in \bar{R}\}) \\
&= \bigsqcup\{\kappa^{-1}(\tilde{r}) \mid \tilde{r} \in \bigvee\{\kappa(\bar{r}) \mid \bar{r} \in \bar{R}\}\} \\
&= \bigsqcup\{\kappa^{-1}(\kappa(\bar{r})) \mid \bar{r} \in \bar{R}\} \\
&= \bigsqcup\{\bar{r} \mid \bar{r} \in \bar{R}\} \\
&= \bar{R}
\end{aligned}$$

$(\kappa \circ \kappa^{-1})$ is the identity for $\mathbf{A}$. Consider $\tilde{R} \in \mathbf{A}$:

$$\begin{aligned}
(\kappa \circ \kappa^{-1})(\tilde{R}) &= \kappa(\bigsqcup\{\kappa^{-1}(\tilde{r}) \mid \tilde{r} \in \tilde{R}\}) \\
&= \bigvee\{\kappa(\bar{r}) \mid \bar{r} \in \bigsqcup\{\kappa^{-1}(\tilde{r}) \mid \tilde{r} \in \tilde{R}\}\} \\
&= \bigvee\{\kappa(\kappa^{-1}(\tilde{r})) \mid \tilde{r} \in \tilde{R}\} \\
&= \bigvee\{\tilde{r} \mid \tilde{r} \in \tilde{R}\} \\
&= \tilde{R}
\end{aligned}$$

4.A.3 Proofs of Section 4.3

Lemma 4.3.2. *Given a concrete sequence r and a concrete constraint $c \in \mathcal{C}$, the abstract operator of propagation $\tilde{\downarrow}$ is such that $\tilde{\alpha}(r\downarrow_c) = \tilde{\alpha}(r)\tilde{\downarrow}_{\tau^+(c)}$*

Proof.

We proceed by induction on the structure of r.

$\underline{\boldsymbol{r = \epsilon}}\lrcorner$ $\forall c \in \mathbf{C}$. $\tilde{\alpha}(\epsilon\downarrow_c) = \epsilon = \tilde{\alpha}(\epsilon)\tilde{\downarrow}_{\tau^+(c)}$.

$\underline{\boldsymbol{r = \boxtimes}}\lrcorner$ $\forall c \in \mathbf{C}$. $\tilde{\alpha}(\boxtimes\downarrow_c) = \boxtimes = \tilde{\alpha}(\boxtimes)\tilde{\downarrow}_{\tau^+(c)}$.

$\underline{\boldsymbol{r = (\eta^+, \eta^-) \rightarrowtail c \cdot r'}}\lrcorner$ In case $c \not\gg (\eta^+, \eta^-)$, $r\downarrow_c$ and $\tilde{\alpha}(r\downarrow_c)$ are not defined. From Equation (4.2.3), it follows that $\tau^+(c) \mathrel{\tilde{\not\gg}} (\tau^+(\eta^+), \tau^-(\eta^-))$, thus, neither $\tilde{\alpha}(r)\tilde{\downarrow}_{\tau^+(c)}$ is defined. Otherwise, if $c \gg (\eta^+, \eta^-)$ we can distinguish two cases.

$\underline{\boldsymbol{c \otimes a \neq \mathit{false}}}\lrcorner$ By Equation (4.2.23), $\tilde{\alpha}(r) = [(\tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(a)]^n \cdot \tilde{\alpha}(r'')$ and $\tilde{\alpha}(r') = [(\tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(a)]^{n-1} \cdot \tilde{\alpha}(r'')$ with $n \geq 1$. Moreover, from Equation (4.2.3), $c \gg (\eta^+, \eta^-) \Rightarrow \tau^+(c) \mathrel{\tilde{\gg}} (\tau^+(\eta^+), \tau^-(\eta^-))$ and, from Equation (4.2.1c), $c \otimes a \neq \mathit{false} \Rightarrow \tau^+(a) \mathbin{\hat{\otimes}} \tau^+(c) \neq \widehat{\mathit{false}}$.

$$\begin{aligned}
\tilde{\alpha}(r\downarrow_c) &= \tilde{\alpha}(((\eta^+, \eta^-) \rightarrowtail a \cdot r')\downarrow_c) \\
&\qquad [\text{since } c \gg (\eta^+, \eta^-) \text{ and } c \otimes a \neq \mathit{false}\,] \\
&= \tilde{\alpha}((c \otimes \eta^+, \eta^-) \rightarrowtail c \otimes a \cdot (r'\downarrow_c)) \\
&\qquad [\text{by } (4.2.23)\,] \\
&= \kappa([(\tau^+(c \otimes \eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(c \otimes a)]^1 \cdot \tilde{\alpha}(r'\downarrow_c)) \\
&\qquad [\text{by Inductive Hypothesis}\,] \\
&= \kappa([(\tau^+(c \otimes \eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(c \otimes a)]^1 \cdot \tilde{\alpha}(r')\tilde{\downarrow}_{\tau^+(c)}) \\
&\qquad [\text{by } (4.2.1\text{c})\,] \\
&= \kappa([(\tau^+(c) \mathbin{\hat{\otimes}} \tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(c) \mathbin{\hat{\otimes}} \tau^+(a)]^1 \cdot \tilde{\alpha}(r')\tilde{\downarrow}_{\tau^+(c)}) \\
&\qquad [\text{since } \tau^+(c) \mathrel{\tilde{\gg}} (\tau^+(\eta^+), \tau^-(\eta^-)) \text{ and } \tau^+(a) \mathbin{\hat{\otimes}} \tau^+(c) \neq \widehat{\mathit{false}}\,] \\
&= \kappa([(\tau^+(c) \mathbin{\hat{\otimes}} \tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(c) \mathbin{\hat{\otimes}} \tau^+(a)]^1 \cdot \\
&\qquad [(\tau^+(c) \mathbin{\hat{\otimes}} \tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(c) \mathbin{\hat{\otimes}} \tau^+(a)]^{n-1} \cdot \tilde{\alpha}(r'')\tilde{\downarrow}_{\tau^+(c)}) \\
&\qquad [\text{by } (4.2.14)\,] \\
&= [(\tau^+(c) \mathbin{\hat{\otimes}} \tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(c) \mathbin{\hat{\otimes}} \tau^+(a)]^n \cdot \tilde{\alpha}(r'')\tilde{\downarrow}_{\tau^+(c)} \\
&\qquad [\text{by Definition 4.3.1}\,] \\
&= \tilde{\alpha}(r)\tilde{\downarrow}_{\tau^+(c)}
\end{aligned}$$

$\underline{\boldsymbol{c \otimes a = \mathit{false}}}\lrcorner$ Similarly to the previous case, by Equation (4.2.3), $c \gg (\eta^+, \eta^-) \Rightarrow \tau^+(c) \mathrel{\tilde{\gg}} (\tau^+(\eta^+), \tau^-(\eta^-))$ and by Equation (4.2.1c), $c \otimes a = \mathit{false} \Rightarrow \tau^+(a) \mathbin{\hat{\otimes}} \tau^+(c) = \widehat{\mathit{false}}$.

$$\begin{aligned}
\tilde{\alpha}(r\downarrow_c) &= \tilde{\alpha}(((\eta^+, \eta^-) \rightarrowtail a \cdot r')\downarrow_c) \\
&\qquad [\text{since } c \gg (\eta^+, \eta^-) \text{ and } c \otimes a = \mathit{false}\,]
\end{aligned}$$

$$
\begin{aligned}
&= \tilde{\alpha}((c \otimes \eta^+, \eta^-) \rightarrowtail \mathit{false} \cdot \boxtimes)\\
&= [(\tau^+(c \otimes \eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(c \otimes a)]^1\\
&\qquad [\text{by Equation (4.2.1c)}]\\
&= [(\tau^+(c) \mathbin{\hat{\otimes}} \tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(c) \mathbin{\hat{\otimes}} \tau^+(a)]^1\\
&\qquad [\text{by Definition 4.3.1}]\\
&= \tilde{\alpha}(r) \tilde{\downarrow}_{\tau^+(c)}
\end{aligned}
$$

$\underline{\boldsymbol{r = stutt(\eta^-) \cdot r'}}$ In case it exists $c^- \in \eta^-$ such that $c \vdash c^-$, $r{\downarrow_c}$ and, consequently, $\tilde{\alpha}(r{\downarrow_c})$ are not defined. From Equation (4.2.1i) it follows that $\tau^+(c) \mathrel{\tilde{\vdash}} \tau^-(\eta^-)$, thus neither $\tilde{\alpha}(r)\tilde{\downarrow}_{\tau^+(c)}$ is defined.

Otherwise, if $\forall h^- \in \eta^- . c \nvdash h^-$ we have that $\tilde{\alpha}(stutt(\eta^-) \cdot r') = [stutt(\tau^-(\eta^-))]^n \cdot \tilde{\alpha}(r'')$ and $\tilde{\alpha}(stutt(r')) = [stutt(\tau^-(\eta^-))]^{n-1} \cdot \tilde{\alpha}(r'')$ with $n \geq 1$

$$
\begin{aligned}
\tilde{\alpha}(r{\downarrow_c}) &= \tilde{\alpha}(stutt(\eta^-) \cdot r'{\downarrow_c})\\
&\qquad [\text{by (4.2.23)}]\\
&= \kappa([stutt(\tau^-(\eta^-))]^1 \cdot \tilde{\alpha}(r'{\downarrow_c}))\\
&\qquad [\text{by Inducctive Hypothesis}]\\
&= \kappa([stutt(\tau^-(\eta^-))]^1 \cdot \tilde{\alpha}(r')\tilde{\downarrow}_{\tau^+(c)})\\
&= \kappa([stutt(\tau^-(\eta^-))]^1 \cdot [stutt(\tau^-(\eta^-))]^{n-1} \cdot \tilde{\alpha}(r''){\downarrow}_{\tau^+(c)})\\
&\qquad [\text{by (4.2.14)}]\\
&= [stutt(\tau^-(\eta^-))]^n \cdot \tilde{\alpha}(r''){\downarrow}_{\tau^+(c)}\\
&\qquad [\text{by Definition 4.3.1}]\\
&= \tilde{\alpha}(r)\tilde{\downarrow}_{\tau^+(c)}
\end{aligned}
$$

Lemma 4.3.6. *Let $r_1, r_2 \in \mathbf{M}$, then $\tilde{\alpha}(r_1 \mathbin{\bar{\|}} r_2) = \tilde{\alpha}(r_1) \mathbin{\tilde{\|}} \tilde{\alpha}(r_2)$ holds.*

Proof.

We proceed by induction on the structure of r_1 [4].

$\underline{\boldsymbol{r_1 = \epsilon} \textbf{ and any } \boldsymbol{r_2}}$ $\tilde{\alpha}(r_1 \mathbin{\bar{\|}} r_2) = \tilde{\alpha}(r_2) = \epsilon \mathbin{\tilde{\|}} \tilde{\alpha}(r_2) = \tilde{\alpha}(r_1) \mathbin{\tilde{\|}} \tilde{\alpha}(r_2)$.

$\underline{\boldsymbol{r_1 = \boxtimes} \textbf{ and any } \boldsymbol{r_2}}$ $\tilde{\alpha}(r_1 \mathbin{\bar{\|}} r_2) = \tilde{\alpha}(r_2) = \boxtimes \mathbin{\tilde{\|}} \tilde{\alpha}(r_2) = \tilde{\alpha}(r_1) \mathbin{\tilde{\|}} \tilde{\alpha}(r_2)$.

$\underline{\boldsymbol{r_1 = \eta_1 \rightarrowtail c_1 \cdot r'_1} \textbf{ and } \boldsymbol{r_2 = \eta_2 \rightarrowtail c_2 \cdot r'_2}}$ In case $\eta_1 \otimes \eta_2$ is not a valid condition $r_1 \mathbin{\bar{\|}} s_2$ is not defined and, as a consequence, $\tilde{\alpha}(r_1 \mathbin{\bar{\|}} s_2)$ is not defined. Moreover, by 4.2.2 it follows that $(\tau^+(\eta_1^+) \mathbin{\hat{\otimes}} \tau^+(\eta_2^+), \tau^-(\eta_1^-) \mathbin{\check{\oplus}} \tau^-(\eta_2^-))$ is abstractly invalid, thus, neither $\tilde{\alpha}(r_1) \mathbin{\tilde{\|}} \tilde{\alpha}(r_2)$ is defined in this case.

Otherwise, if $\eta_1 \otimes \eta_2$ is a valid condition, by Equation (4.2.2) its abstraction is valid too, thus, we can distinguish two cases.

[4]The cases for r_2 are symmetric.

$\underline{\boldsymbol{c_1 \otimes c_2 \neq \mathit{false}}}\lrcorner$ From 4.2.1c, it follows that $\tau^+(c_1) \hat{\otimes} \tau^+(c_2) \neq \widehat{\mathit{false}}$.

$$
\begin{aligned}
&\tilde{\alpha}(r_1 \mathbin{\bar{\|}} s_2) = \\
&= \kappa(\alpha^{\pm}((\eta_1 \otimes \eta_2) \rightarrowtail c_1 \otimes c_2 \cdot (r_1'{\downarrow}_{c_2} \mathbin{\bar{\|}} r_2'{\downarrow}_{c_1}))) \\
&\qquad [\,\text{by (4.2.5) and (4.2.2)}\,] \\
&= \kappa([(\tau^+(\eta_1^+ \otimes \eta_2^+), \tau^-(\eta_1^- \cup \eta_2^-)) \rightarrowtail \tau^+(c_1 \otimes c_2)]^1 \cdot \tilde{\alpha}(r_1'{\downarrow}_{c_2} \mathbin{\bar{\|}} r_2'{\downarrow}_{c_1})) \\
&\qquad [\,\text{by properties 4.2.1c } (\tau^+(c_1) \hat{\otimes} \tau^+(c_2) \neq \widehat{\mathit{false}}) \text{ and 4.2.1d}\,] \\
&= \kappa(\,[(\tau^+(\eta_1^+) \hat{\otimes} \tau^+(\eta_2^+), \tau^-(\eta_1^-) \check{\oplus} \tau^-(\eta_2^-)) \rightarrowtail \tau^+(c_1) \hat{\otimes} \tau^+(c_2)]^1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \cdot \tilde{\alpha}(r_1'{\downarrow}_{c_2} \mathbin{\bar{\|}} r_2'{\downarrow}_{c_1})) \\
&\qquad [\,\text{by Inductive Hypothesis}\,] \\
&= \kappa(\,[(\tau^+(\eta_1^+) \hat{\otimes} \tau^+(\eta_2^+), \tau^-(\eta_1^-) \check{\oplus} \tau^-(\eta_2^-)) \rightarrowtail \tau^+(c_1) \hat{\otimes} \tau^+(c_2)]^1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \cdot \tilde{\alpha}(r_1'{\downarrow}_{c_2}) \mathbin{\tilde{\|}} \tilde{\alpha}(r_2'{\downarrow}_{c_1})) \\
&\qquad [\,\text{by Lemma 4.3.2}\,] \\
&= \kappa(\,[(\tau^+(\eta_1^+) \hat{\otimes} \tau^+(\eta_2^+), \tau^-(\eta_1^-) \check{\oplus} \tau^-(\eta_2^-)) \rightarrowtail \tau^+(c_1) \hat{\otimes} \tau^+(c_2)]^1 \\
&\qquad\qquad\qquad\qquad\qquad \cdot \tilde{\alpha}(r_1')\tilde{\downarrow}_{\tau^+(c_2)} \mathbin{\tilde{\|}} \tilde{\alpha}(r_2')\tilde{\downarrow}_{\tau^+(c_1)}) \\
&\qquad [\,\text{by Definition 4.3.5}\,] \\
&= \ \kappa([(\tau^+(\eta_1^+), \tau^-(\eta_1^-)) \rightarrowtail \tau^+(c_1)]^1 \cdot \tilde{\alpha}(r_1')\tilde{\downarrow}_{\tau^+(c_2)}) \\
&\quad\ \tilde{\|} \\
&\quad\ \kappa([(\tau^+(\eta_2^+), \tau^-(\eta_2^-)) \rightarrowtail \tau^+(c_2)]^1 \cdot \tilde{\alpha}(r_2')\tilde{\downarrow}_{\tau^+(c_2)}) \\
&\qquad [\,\text{by (4.2.5)}\,] \\
&= \kappa(\alpha^{\pm}(r_A)) \mathbin{\tilde{\|}} \kappa(\alpha^{\pm}(r_B)) \\
&\qquad [\,\text{by (4.2.23)}\,] \\
&= \tilde{\alpha}(r_1) \mathbin{\tilde{\|}} \tilde{\alpha}(r_2)
\end{aligned}
$$

$\underline{\boldsymbol{c_1 \otimes c_2 = \mathit{false}}}\lrcorner$

$$
\begin{aligned}
&\tilde{\alpha}(r_1 \mathbin{\bar{\|}} s_2) = \\
&\quad = \kappa(\alpha^{\pm}((\eta \otimes \delta) \rightarrowtail \mathit{false} \cdot \boxtimes)) \\
&\qquad [\,\text{by (4.2.5), (4.2.2) and since } \tau^+(c_1) \hat{\otimes} \tau^+(c_2) \neq \widehat{\mathit{false}}\,] \\
&\quad = [(\tau^+(\eta^+ \otimes \delta^+), \tau^-(\eta^- \cup \delta^-)) \rightarrowtail \widehat{\mathit{false}}]^1 \cdot \boxtimes \\
&\qquad [\,\text{by properties 4.2.1c and 4.2.1d}\,] \\
&\quad = [(\tau^+(\eta^+) \hat{\otimes} \tau^+(\delta^+), \tau^-(\eta^-) \check{\oplus} \tau^-(\delta^-)) \rightarrowtail \widehat{\mathit{false}}]^1 \\
&\qquad [\,\text{by Definition 4.3.5 and since } \tau^+(a) \hat{\otimes} \tau^+(b) = \widehat{\mathit{false}}\,] \\
&\quad = \ \kappa([(\tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(a)]^1 \cdot \alpha^{\pm}(r_A')) \\
&\qquad \tilde{\|} \\
&\qquad \kappa([(\tau^+(\delta^+), \tau^-(\delta^-)) \rightarrowtail \tau^+(b)]^1 \cdot \alpha^{\pm}(r_B')) \\
&\qquad [\,\text{by (4.2.5)}\,]
\end{aligned}
$$

$$
\begin{aligned}
&= \kappa(\alpha^{\pm}(r_1)) \mathbin{\tilde{\parallel}} \kappa(\alpha^{\pm}(r_2)) \\
&\qquad [\,\text{by } (4.2.23)\,] \\
&= \tilde{\alpha}(r_1) \mathbin{\tilde{\parallel}} \tilde{\alpha}(r_2)
\end{aligned}
$$

$r_1 = \eta_1 \rightarrowtail c_1 \cdot r_1'$ and $r_2 = stutt(\eta_2^-) \cdot r_2'$. In case $(\eta_1^+, \eta_1^- \cup \eta_2^-)$ is not a valid condition $r_1 \mathbin{\bar{\parallel}} s_2$ is not defined and, as a consequence, $\tilde{\alpha}(r_1 \mathbin{\bar{\parallel}} s_2)$ is not defined. Moreover, by 4.2.2 it follows that $(\tau^+(\eta_1^+), \tau^-(\eta_1^-) \mathbin{\check{\oplus}} \tau^-(\eta_2^-))$ is abstractly invalid, thus, neither $\tilde{\alpha}(r_1) \mathbin{\tilde{\parallel}} \tilde{\alpha}(r_2)$ is defined in this case. Otherwise, if $(\eta_1^+, \eta_1^- \cup \eta_2^-)$ is a valid condition, by Equation (4.2.2) its abstraction is valid too, thus, we can distinguish two cases.

$$
\begin{aligned}
&\tilde{\alpha}(r_1 \mathbin{\bar{\parallel}} s_2) = \\
&\quad = \kappa(\alpha^{\pm}((\eta_1^+, \eta_1^- \cup \eta_2^-) \rightarrowtail c_1 \cdot r_1' \mathbin{\bar{\parallel}} r_2'{\downarrow_{c_1}})) \\
&\qquad [\,\text{by } (4.2.5) \text{ and } (4.2.2)\,] \\
&\quad = \kappa([(\tau^+(\eta_1^+), \tau^-(\eta_1^- \cup \eta_2^-)) \rightarrowtail \tau^+(c_1)]^1 \cdot \tilde{\alpha}(r_1' \mathbin{\bar{\parallel}} r_2'{\downarrow_{c_1}})) \\
&\qquad [\,\text{by } (4.2.1\text{d})\,] \\
&\quad = \kappa([(\tau^+(\eta_1^+), \tau^-(\eta_1^-) \mathbin{\check{\oplus}} \tau^-(\eta_2^-)) \rightarrowtail \tau^+(c_1)]^1 \cdot \tilde{\alpha}(r_1' \mathbin{\bar{\parallel}} r_2'{\downarrow_{c_1}})) \\
&\qquad [\,\text{by Inductive Hypothesis}\,] \\
&\quad = \kappa([(\tau^+(\eta_1^+), \tau^-(\eta_1^-) \mathbin{\check{\oplus}} \tau^-(\eta_2^-)) \rightarrowtail \tau^+(c_1)]^1 \cdot (\tilde{\alpha}(r_1') \mathbin{\tilde{\parallel}} \tilde{\alpha}(r_2'{\downarrow_{c_1}}))) \\
&\qquad [\,\text{by Lemma 4.3.2}\,] \\
&\quad = \kappa([(\tau^+(\eta_1^+), \tau^-(\eta_1^-) \mathbin{\check{\oplus}} \tau^-(\eta_2^-)) \rightarrowtail \tau^+(c_1)]^1 \cdot (\tilde{\alpha}(r_1') \mathbin{\tilde{\parallel}} \tilde{\alpha}(r_2')\tilde{\downarrow}_{\tau^+(c_1)})) \\
&\qquad [\,\text{by Definition 4.3.5}\,] \\
&\quad = \kappa([(\tau^+(\eta_1^+), \tau^-(\eta_1^-)) \rightarrowtail \tau^+(c_1)]^1 \cdot \tilde{\alpha}(r_1')) \mathbin{\tilde{\parallel}} \kappa([stutt(\tau^-(\eta_2^-))]^1 \cdot \tilde{\alpha}(r_2')\tilde{\downarrow}_{\tau^+(c_1)}) \\
&\qquad [\,\text{by } (4.2.5)\,] \\
&\quad = \kappa(\alpha^{\pm}(r_1)) \mathbin{\tilde{\parallel}} \kappa(\alpha^{\pm}(r_2)) \\
&\qquad [\,\text{by } (4.2.23)\,] \\
&\quad = \tilde{\alpha}(r_1) \mathbin{\tilde{\parallel}} \tilde{\alpha}(r_2)
\end{aligned}
$$

$r_1 = stutt(\eta_1^-) \cdot r_1'$ and $r_2 = stutt(\eta_2^-) \cdot r_2'$.

$$
\begin{aligned}
&\tilde{\alpha}(r_1 \mathbin{\bar{\parallel}} s_2) = \\
&\quad = \kappa(\alpha^{\pm}(stutt(\eta_1^- \cup \eta_2^-) \cdot r_1' \mathbin{\bar{\parallel}} r_2')) \\
&\qquad [\,\text{by } (4.2.5)\,] \\
&\quad = \kappa([stutt(\tau^-(\eta_1^- \cup \eta_2^-))]^1 \cdot \tilde{\alpha}(r_1' \mathbin{\bar{\parallel}} r_2')) \\
&\qquad [\,\text{by } (4.2.1\text{d})\,] \\
&\quad = \kappa([stutt(\tau^-(\eta_1^-) \mathbin{\check{\oplus}} \tau^-(\eta_2^-))]^1 \cdot \tilde{\alpha}(r_1' \mathbin{\bar{\parallel}} r_2')) \\
&\qquad [\,\text{by Inductive Hypothesis}\,] \\
&\quad = \kappa([stutt(\tau^-(\eta_1^-) \mathbin{\check{\oplus}} \tau^-(\eta_2^-))]^1 \cdot \tilde{\alpha}(r_A') \mathbin{\tilde{\parallel}} \tilde{\alpha}(r_B')) \\
&\qquad [\,\text{by Definition 4.3.5}\,] \\
&\quad = \kappa([stutt(\tau^-(\eta_1^-))]^1 \cdot \tilde{\alpha}(r_1')) \mathbin{\tilde{\parallel}} \kappa([stutt(\tau^-(\eta_1^-))]^1 \cdot \tilde{\alpha}(r_2')) \\
&\qquad [\,\text{by } (4.2.5)\,]
\end{aligned}
$$

$$
\begin{aligned}
&= \kappa(\alpha^{\pm}(r_1)) \mathbin{\tilde{\|}} \kappa(\alpha^{\pm}(r_2)) \\
&\qquad [\,\text{by (4.2.23)}\,] \\
&= \tilde{\alpha}(r_1) \mathbin{\tilde{\|}} \tilde{\alpha}(r_2)
\end{aligned}
$$

Lemma 4.3.8. *Given $r \in \mathbf{M}$ and $x \in Var$, $\tilde{\alpha}(\bar{\exists}_x r) = \kappa(\tilde{\exists}_x \tilde{\alpha}(r))$.*

Proof.

We proceed by structural induction on r.

$\boldsymbol{r = \epsilon}$ **or** $\boldsymbol{r = \boxtimes}$ In this case the statement follows directly from Definition 4.3.7.

$\boldsymbol{r = (\eta^+, \eta^-) \rightarrowtail c \cdot r'}$

$$
\begin{aligned}
\tilde{\alpha}(\bar{\exists}_x r) &= \tilde{\alpha}(\bar{\exists}_x((\eta^+, \eta^-) \rightarrowtail a \cdot r')) \\
&\qquad [\,\text{by (4.2.23)}\,] \\
&= \kappa(\alpha^{\pm}((\exists_x \eta^+, \exists_x \eta^-) \rightarrowtail \exists_x a \cdot \bar{\exists}_x r')) \\
&\qquad [\,\text{by (4.2.5)}\,] \\
&= \kappa([(\tau^+(\exists_x \eta^+), \tau^-(\exists_x \eta^-)) \rightarrowtail \tau^+(\exists_x a)]^1 \cdot \alpha^{\pm}(\bar{\exists}_x r')) \\
&\qquad [\,\text{by (4.2.1g) and (4.2.1h)}\,] \\
&= \kappa([(\hat{\exists}_x \tau^+(\eta^+), \check{\exists}_x \tau^-(\eta^-)) \rightarrowtail \hat{\exists}_x \tau^+(a)]^1 \cdot \alpha^{\pm}(\bar{\exists}_x r')) \\
&\qquad [\,\text{by Inductive Hypothesis}\,] \\
&= \kappa([(\hat{\exists}_x \tau^+(\eta^+), \check{\exists}_x \tau^-(\eta^-)) \rightarrowtail \hat{\exists}_x \tau^+(a)]^1 \cdot \tilde{\exists}_x \alpha^{\pm}(r')) \\
&\qquad [\,\text{by (4.2.5)}\,] \\
&= \kappa(\tilde{\exists}_x \alpha^{\pm}(r)) \\
&\qquad [\,\text{since } \kappa \text{ is idempotent}\,] \\
&= \kappa(\tilde{\exists}_x \kappa(\alpha^{\pm}(r))) \\
&\qquad [\,\text{by (4.2.23)}\,] \\
&= \kappa(\tilde{\exists}_x \tilde{\alpha}(r))
\end{aligned}
$$

$\boldsymbol{r = stutt(\eta^-) \cdot r'}$

$$
\begin{aligned}
\tilde{\alpha}(\bar{\exists}_x r) &= \tilde{\alpha}(\bar{\exists}_x(stutt(\eta^-) \cdot r')) \\
&\qquad [\,\text{by (4.2.23)}\,] \\
&= \kappa(\alpha^{\pm}(stutt(\exists_x \eta^-) \cdot \bar{\exists}_x r')) \\
&= \kappa([stutt(\tau^-(\exists_x \eta^-))]^1 \cdot \alpha^{\pm}(\bar{\exists}_x r')) \\
&\qquad [\,\text{by (4.2.1h)}\,] \\
&= \kappa([stutt(\check{\exists}_x \tau^-(\eta^-))]^1 \cdot \alpha^{\pm}(\bar{\exists}_x r')) \\
&\qquad [\,\text{by Inductive Hypothesis}\,] \\
&= \kappa([stutt(\check{\exists}_x \tau^-(\eta^-))]^1 \cdot \tilde{\exists}_x \alpha^{\pm}(r')) \\
&\qquad [\,\text{by (4.2.5)}\,]
\end{aligned}
$$

$$
\begin{aligned}
&= \kappa(\tilde{\exists}_x\, \alpha^{\pm}(r)) \\
&\qquad [\,\text{since } \kappa \text{ is idempotent}\,] \\
&= \kappa(\tilde{\exists}_x\, \kappa(\alpha^{\pm}(r))) \\
&\qquad [\,\text{by } (4.2.23)\,] \\
&= \kappa(\tilde{\exists}_x\, \tilde{\alpha}(r))
\end{aligned}
$$

Theorem 4.A.1 *Given an interpretation* $\mathcal{I}^{\pm}$*,*

$$(\tilde{\alpha} \circ \mathcal{D}[\![D]\!] \circ \tilde{\gamma})(\mathcal{I}^{\pm}) = \lambda p(x). \bigvee\nolimits_{p(x):-A \in D} \mathcal{A}^{\pm}[\![A]\!]_{\mathcal{I}^{\pm}}$$

Proof.

$$
\begin{aligned}
(\tilde{\alpha} \circ \mathcal{D}[\![D]\!] \circ \tilde{\gamma})(\mathcal{I}^{\pm}) &= \lambda p(x).\, \tilde{\alpha}(\textstyle\bigsqcup_{p(x):-A\in D} \mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}) \\
&= \lambda p(x). \textstyle\bigvee_{p(x):-A \in D} \tilde{\alpha}(\mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})})\}
\end{aligned}
$$

Now, we show that given an agent A, $\tilde{\alpha}(\mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}) = \mathcal{A}^{\pm}[\![A]\!]_{\mathcal{I}^{\alpha}}$. We proceed by induction on the structure of the agent A.

$\underline{A = \mathsf{skip}}$ In this case the proof is straightforward: $\tilde{\alpha}(\mathcal{A}[\![\mathsf{skip}]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}) = \boxtimes = \mathcal{A}^{\pm}[\![\mathsf{skip}]\!]_{\mathcal{I}^{\pm}}$

$\underline{A = \mathsf{tell}(c)}$

$$
\begin{aligned}
\tilde{\alpha}(\mathcal{A}[\![\mathsf{tell}(c)]\!]_{\gamma(\mathcal{I}^{\alpha})}) &= \tilde{\alpha}(\{(\mathit{true}, \varnothing) \rightarrowtail c \cdot \boxtimes\}) \\
&\qquad [\,\text{by } (4.2.23)\,] \\
&= [(\tau^{+}(\mathit{true}), \tau^{-}(\varnothing)) \rightarrowtail \tau^{+}(c)]^{1} \cdot \boxtimes \\
&\qquad [\,\text{since } \tau^{+}(\mathit{true}) = \hat{\mathit{true}} \text{ and } \tau^{-}(\varnothing) = \check{\mathit{false}}\,] \\
&= [(\hat{\mathit{true}}, \check{\mathit{false}}) \rightarrowtail \tau^{+}(c)]^{1} \cdot \boxtimes \\
&\qquad [\,\text{by } (4.3.4)\,] \\
&= \mathcal{A}^{\pm}[\![\mathsf{tell}(c)]\!]_{\mathcal{I}^{\alpha}}
\end{aligned}
$$

$\underline{A = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i}$ For the sake of brevity, we call F (respectively $\tilde{F}$) the body of the least fixed point in (3.1.7) (respectively (4.3.9)).

$$
\begin{aligned}
F :=& \lambda R.\, (\mathit{stutt}(\{c_1, \dots, c_n\}) \cdot R \sqcup \\
&\quad \textstyle\bigsqcup\{(c_i, \varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}}) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\mathcal{I}},\, r \text{ being } c_i\text{-compatible}\}) \\
\tilde{F} :=& \lambda \tilde{R}.\, (\kappa(\mathit{stutt}(\tau^{-}(\{c_1, \dots, c_n\})) \cdot \tilde{R}) \vee \\
&\quad \textstyle\bigvee\{\kappa([(\tau^{+}(c_i), \check{\mathit{false}}) \rightarrowtail \tau^{+}(c_i)]^{1} \cdot (\tilde{r}\tilde{\downarrow}_{\tau^{+}(c_i)})) \mid 1 \le i \le n,\, \tilde{r} \in \mathcal{A}^{\pm}[\![A_i]\!]_{\mathcal{I}^{\pm}}, \\
&\qquad \tilde{r} \text{ being } \tau^{+}(c_i)\text{-compatible}\})
\end{aligned}
$$

To show that $\mathcal{A}^{\pm}[\![\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}^{\pm}} = \tilde{\alpha}(\mathcal{A}[\![\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})})$ it is necessary to prove first that $\tilde{\alpha} \circ F \circ \tilde{\gamma} \circ \tilde{\alpha} = \tilde{\alpha} \circ F$ and $\tilde{F} = \tilde{\alpha} \circ F \circ \tilde{\gamma}$.

$$
\begin{aligned}
&\tilde{\alpha} \circ F \circ \tilde{\gamma} \circ \tilde{\alpha}\\
&= \tilde{\alpha}(\lambda R.\, (\mathit{stutt}(\{c_1, \dots, c_n\}) \cdot \tilde{\gamma}(\tilde{\alpha}(R)) \sqcup\\
&\qquad \textstyle\bigsqcup\{(c_i, \varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}}) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\mathcal{I}},\, r\ c_i\text{-compatible}\}))\\
&\qquad [\,\text{by the additivity of } \tilde{\alpha}\,]\\
&= \lambda R.\, (\tilde{\alpha}(\mathit{stutt}(\{c_1, \dots, c_n\}) \cdot \tilde{\gamma}(\tilde{\alpha}(R))) \vee\\
&\qquad \textstyle\bigvee \tilde{\alpha}(\{(c_i, \varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}}) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\mathcal{I}},\, r\ c_i\text{-compatible}\}))\\
&\qquad [\,\text{by Definition of } \tilde{\alpha} \text{ (4.2.23) and since } \kappa \text{ is idempotent}\,]\\
&= \lambda R.\, (\kappa(\alpha^{\pm}(\mathit{stutt}(\{c_1, \dots, c_n\}))) \cdot \tilde{\alpha}(\tilde{\gamma}(\tilde{\alpha}(R)))) \vee\\
&\qquad \textstyle\bigvee \tilde{\alpha}(\{(c_i, \varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}}) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\mathcal{I}},\, r\ c_i\text{-compatible}\}))\\
&\qquad [\,\tilde{\alpha} \circ \tilde{\gamma} \text{ is the identity for } \tilde{\mathbf{M}}^{\pm}\,]\\
&= \lambda R.\, (\kappa(\alpha^{\pm}(\mathit{stutt}(\{c_1, \dots, c_n\}))) \cdot \tilde{\alpha}(R)) \vee\\
&\qquad \textstyle\bigvee \tilde{\alpha}(\{(c_i, \varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}}) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\mathcal{I}},\, r\ c_i\text{-compatible}\}))\\
&= \tilde{\alpha}(\lambda R.\, (\mathit{stutt}(\{c_1, \dots, c_n\}) \cdot R \sqcup\\
&\qquad \textstyle\bigsqcup\{(c_i, \varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}}) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\mathcal{I}},\, r\ c_i\text{-compatible}\}))\\
&\qquad [\,\text{by Definition of } \tilde{\alpha} \text{ (4.2.23)}\,]\\
&= \tilde{\alpha} \circ F
\end{aligned}
$$

$$
\begin{aligned}
&\tilde{\alpha} \circ F \circ \tilde{\gamma}\\
&= \tilde{\alpha}(\lambda \tilde{R}.\, (\mathit{stutt}(\{c_1, \dots, c_n\}) \cdot \tilde{\gamma}(\tilde{R}) \sqcup\\
&\qquad \textstyle\bigsqcup\{(c_i, \varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}}) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\, r\ c_i\text{-compatible}\}))\\
&\qquad [\,\text{by the additivity of } \tilde{\alpha}\,]\\
&= \lambda \tilde{R}.\, (\tilde{\alpha}(\mathit{stutt}(\{c_1, \dots, c_n\}) \cdot \tilde{\gamma}(\tilde{R})) \vee\\
&\qquad \textstyle\bigvee \tilde{\alpha}(\{(c_i, \varnothing) \rightarrowtail c_i \cdot (r{\downarrow_{c_i}}) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\, r\ c_i\text{-comp.}\}))\\
&\qquad [\,\text{by Definition of } \tilde{\alpha} \text{ (4.2.23) and since } \kappa \text{ is idempotent}\,]\\
&= \lambda \tilde{R}.\, (\kappa([\mathit{stutt}(\tau^{-}(\{c_1, \dots, c_n\}))]^1 \cdot \tilde{\alpha}(\tilde{\gamma}(\tilde{R}))) \vee\\
&\qquad \textstyle\bigvee\{\kappa([(\tau^{+}(c_i), \tau^{-}(\varnothing)) \rightarrowtail \tau^{+}(c_i)]^1 \cdot \tilde{\alpha}(r{\downarrow_{c_i}})) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\, r\ c_i\text{-comp.}\})\\
&\qquad [\,\tilde{\alpha} \circ \tilde{\gamma} \text{ is the identity for } \tilde{\mathbf{M}}^{\pm}\,]\\
&= \lambda \tilde{R}.\, (\kappa([\mathit{stutt}(\tau^{-}(\{c_1, \dots, c_n\}))]^1 \cdot \tilde{R}) \vee\\
&\qquad \textstyle\bigvee\{\kappa([(\tau^{+}(c_i), \tau^{-}(\varnothing)) \rightarrowtail \tau^{+}(c_i)]^1 \cdot \tilde{\alpha}(r{\downarrow_{c_i}})) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\, r\ c_i\text{-comp.}\})\\
&\qquad [\,\text{by Lemma 4.3.2 and since } \tau^{-}(\varnothing) = \check{\mathit{false}}\,]\\
&= \lambda \tilde{R}.\, (\kappa([\mathit{stutt}(\tau^{-}(\{c_1, \dots, c_n\}))]^1 \cdot \tilde{R}) \vee\\
&\qquad \textstyle\bigvee\{\kappa([(\tau^{+}(c_i), \check{\mathit{false}}) \rightarrowtail \tau^{+}(c_i)]^1 \cdot \tilde{\alpha}(r)\tilde{\downarrow}_{\tau^{+}(c_i)}) \mid 1 \le i \le n,\, r \in \mathcal{A}[\![A_i]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\, r\ c_i\text{-comp.}\})\\
&\qquad [\,\text{by Proposition 4.3.4}\,]
\end{aligned}
$$

$$= \lambda \tilde{R}.\, (\kappa([\mathit{stutt}(\tau^-(\{c_1,\dots,c_n\}))]^1 \cdot \tilde{R}) \vee$$
$$\bigvee\{\kappa([(\tau^+(c_i), \check{\mathit{false}}) \rightarrowtail \tau^+(c_i)]^1 \cdot \tilde{r}\tilde{\downarrow}_{\tau^+(c_i)}) \mid 1 \leq i \leq n,\ \tilde{r} \in \tilde{\alpha}(\mathcal{A}[\![A_i]\!]_{\tilde{\gamma}(\mathcal{I}^\pm)}),\ \tilde{r}\ \tau^+(c_i)\text{-comp.}\})$$
$$[\text{by Inductive Hypothesis}]$$
$$= \lambda \tilde{R}.\, (\kappa([\mathit{stutt}(\tau^-(\{c_1,\dots,c_n\}))]^1 \cdot \tilde{R}) \vee$$
$$\bigvee\{\kappa([(\tau^+(c_i), \check{\mathit{false}}) \rightarrowtail \tau^+(c_i)]^1 \cdot \tilde{r}\tilde{\downarrow}_{\tau^+(c_i)}) \mid 1 \leq i \leq n,\ \tilde{r} \in \mathcal{A}^\pm[\![A_i]\!]_{\mathcal{I}^\pm},\ \tilde{r}\ \tau^+(c_i)\text{-comp.}\})$$
$$= \tilde{F}$$

Now, we can show that for each $n \in \mathbf{N}$ $\tilde{F}{\uparrow}n = \tilde{\alpha}(F{\uparrow}n)$.

$$\begin{aligned}
\tilde{F}{\uparrow}n &= (\tilde{\alpha} \circ F \circ \tilde{\gamma})^n(\{\epsilon\}) \\
&= (\tilde{\alpha} \circ F \circ \tilde{\gamma} \circ \tilde{\alpha} \circ F \circ \tilde{\gamma} \dots \tilde{\alpha} \circ F \circ \tilde{\gamma})(\{\epsilon\}) \\
&\qquad [\,\tilde{\alpha} \circ F \circ \tilde{\gamma} \circ \tilde{\alpha} = \tilde{\alpha} \circ F\,] \\
&= (\tilde{\alpha} \circ F^n \circ \tilde{\gamma}())(\{\epsilon\}) \\
&\qquad [\,\text{since } \tilde{\gamma}(\{\epsilon\}) = \{\epsilon\}\,] \\
&= (\tilde{\alpha} \circ F^n)(\{\epsilon\}) \\
&= \tilde{\alpha}(F{\uparrow}n)
\end{aligned}$$

By considering the limit of the iterations, we can conclude that $\mathcal{A}^\pm[\![\sum_{i=1}^n \mathsf{ask}(c_i) \rightarrow A_i]\!]_{\mathcal{I}^\pm} = \tilde{\alpha}(\mathcal{A}[\![\sum_{i=1}^n \mathsf{ask}(c_i) \rightarrow A_i]\!]_{\tilde{\gamma}(\mathcal{I}^\pm)})$.

$$\begin{aligned}
\mathcal{A}^\pm[\![\sum_{i=1}^n \mathsf{ask}(c_i) \rightarrow A_i]\!]_{\mathcal{I}^\pm} &= \mathit{lfp}_{\tilde{\mathbf{M}}^\pm}\, \tilde{F} \\
&= \textstyle\bigvee_{n \geq 0} \tilde{F}{\uparrow}n \\
&\qquad [\,\forall n\ \tilde{F}{\uparrow}n = \tilde{\alpha}(F){\uparrow}n\,] \\
&= \textstyle\bigvee_{n \geq 0} \tilde{\alpha}(F){\uparrow}n \\
&\qquad [\,\tilde{\alpha} \text{ additive}\,] \\
&= \tilde{\alpha}(\textstyle\bigsqcup_{n \geq 0} F{\uparrow}n) \\
&= \tilde{\alpha}(\mathit{lfp}_{\mathbf{M}}\, F) \\
&= \tilde{\alpha}(\mathcal{A}[\![\textstyle\sum_{i=1}^n \mathsf{ask}(c_i) \rightarrow A_i]\!]_{\tilde{\gamma}(\mathcal{I}^\pm)})
\end{aligned}$$

A = now c then A else B

$\tilde{\alpha}(\mathcal{A}[\![\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\tilde{\gamma}(\mathcal{I}^\pm)})$

$$
\begin{aligned}
= \tilde{\alpha}(\ & \{(c,\varnothing) \rightarrowtail c \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\} \sqcup \\
& \bigsqcup\{(\eta^+ \otimes c, \eta^-) \rightarrowtail d \otimes c \cdot (r{\downarrow_c}) \mid (\eta^+, \eta^-) \rightarrowtail d \cdot r \in \mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\ d \otimes c \neq \mathit{false}, \\
& \qquad\qquad\qquad\qquad \forall c^- \in \eta^- .\, \eta^+ \otimes c \nvdash c^-\} \sqcup \\
& \bigsqcup\{(\eta^+ \otimes c, \eta^-) \rightarrowtail \hat{\mathit{false}} \mid (\eta^+, \eta^-) \rightarrowtail d \cdot r \in \mathcal{A}[\![A]\!]_{\gamma(\mathcal{I}^{\alpha})},\ d \otimes c = \mathit{false} \\
& \qquad\qquad\qquad\qquad \forall c^- \in \eta^- .\, \eta^+ \otimes c \nvdash c^-\} \sqcup \\
& \bigsqcup\{(c, \eta^-) \rightarrowtail c \cdot (r{\downarrow_c}) \mid \ \mathit{stutt}(\eta^-) \cdot r \in \mathcal{A}[\![A]\!]_{\gamma(\mathcal{I}^{\alpha})} \forall c^- \in \eta^- .\, \eta^+ \otimes c \nvdash c^-\} \sqcup \\
& \{(\mathit{true}, \{c\}) \rightarrowtail \mathit{true} \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\} \sqcup \\
& \bigsqcup\{(\eta^+, \eta^- \cup \{c\}) \rightarrowtail d \cdot r \mid (\eta^+, \eta^-) \rightarrowtail d \cdot r \in \mathcal{A}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\ \eta^+ \nvdash c\} \sqcup \\
& \bigsqcup\{(\mathit{true}, \eta^- \cup \{c\}) \rightarrowtail \mathit{true} \cdot r \mid \mathit{stutt}(\eta^-) \cdot r \in \mathcal{A}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\})
\end{aligned}
$$

[by Definition of $\tilde{\alpha}$ (4.2.23)]

$$
\begin{aligned}
= \ & \{[(\tau^+(c), \overbrace{\tau^-(\varnothing)}^{\check{\mathit{false}}}) \rightarrowtail \tau^+(c)]^1 \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\} \vee \\
& \bigvee\{\kappa([(\overbrace{\tau^+(\eta^+ \otimes c)}^{c\hat{\times}\tau^+(\eta^+)}, \tau^-(\eta^-)) \rightarrowtail \overbrace{\tau^+(a \otimes c)}^{c\hat{\times}\tau^+(d)}]^1 \cdot \overbrace{\tilde{\alpha}(r{\downarrow_c})}^{\tilde{\alpha}(r)\tilde{\downarrow}_{\tau^+(c)}}) \mid \\
& \qquad (\eta^+, \eta^-) \rightarrowtail d \cdot r \in \mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\ c \otimes a \neq \mathit{false} \text{ and } \forall h^- \in \eta^- .\, c \otimes \eta^+ \nvdash h^-\} \vee \\
& \bigvee\{[(\overbrace{\tau^+(\eta^+ \otimes c)}^{c\hat{\times}\tau^+(\eta^+)}, \tau^-(\eta^-)) \rightarrowtail \overbrace{\tau^+(\mathit{false})}^{\hat{\mathit{false}}}]^1 \mid (\eta^+, \eta^-) \rightarrowtail d \cdot r \in \mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\ c \otimes a = \mathit{false} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{and } \forall h^- \in \eta^- .\, c \otimes \eta^+ \nvdash h^-\} \vee \\
& \bigvee\{\kappa([(\tau^+(c), \tau^-(\eta^-)) \rightarrowtail \tau^+(c)]^1 \cdot \overbrace{\tilde{\alpha}(r{\downarrow_c})}^{\tilde{\alpha}(r)\tilde{\downarrow}_{\tau^+(c)}}) \mid \ \mathit{stutt}(\eta^-) \cdot r \in \mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})} \\
& \qquad\qquad\qquad\qquad\qquad\qquad \forall h^- \in \eta^- .\, \eta^+ \nvdash h^-\} \vee \\
& \{[(\overbrace{\tau^+(\mathit{true})}^{\hat{\mathit{true}}}, \tau^-(\{c\})) \rightarrowtail \overbrace{\tau^+(\mathit{true})}^{\hat{\mathit{true}}}]^1 \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\} \vee \\
& \bigvee\{\kappa([(\tau^+(\eta^+), \overbrace{\eta^- \cup \tau^-(\{c\})}^{c\check{\times}\tau^-(\eta^-)}) \rightarrowtail \tau^+(d)]^1 \cdot \tilde{\alpha}(r)) \mid (\eta^+, \eta^-) \rightarrowtail d \cdot r \in \mathcal{A}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\, c \nvdash \eta^+\} \vee \\
& \bigvee\{\kappa([(\overbrace{\tau^+(\mathit{true})}^{\hat{\mathit{true}}}, \overbrace{\eta^- \cup \tau^-(\{c\})}^{c\check{\times}\tau^-(\eta^-)}) \rightarrowtail \overbrace{\tau^+(\mathit{true})}^{\hat{\mathit{true}}}]^1 \cdot \tilde{\alpha}(r)) \mid \mathit{stutt}(\eta^-) \cdot r \in \mathcal{A}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\}
\end{aligned}
$$

[by Properties 4.2.1 and by Lemma 4.3.2]

$$\begin{aligned}
= {} & \{[(\tau^+(c), \check{\mathit{false}}) \rightarrowtail \tau^+(c)]^1 \cdot \boxtimes \mid \boxtimes \in \tilde{\alpha}(\mathcal{A}\llbracket A \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)})\} \vee \\
& \bigvee\{ \kappa([(c \mathbin{\hat{\times}} \tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail c \mathbin{\hat{\times}} \tau^+(d)]^1 \cdot \alpha^\pm(r)\tilde{\downarrow}_{\tau^+(c)}) \mid c \mathbin{\hat{\times}} \tau^+(d) \neq \hat{\mathit{false}}, \\
& \qquad c \mathbin{\hat{\times}} \tau^+(\eta^+) \mathrel{\tilde{\not\vdash}} \tau^-(\eta^-),\ \kappa([(\tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(d)]^1 \cdot \alpha^\pm(r)) \in \tilde{\alpha}(\mathcal{A}\llbracket A \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)})\} \vee \\
& \bigvee\{[(c \mathbin{\hat{\times}} \tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \hat{\mathit{false}}]^1 \mid \\
& \qquad \kappa([(\tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(d)]^1 \cdot \alpha^\pm(r)) \in \tilde{\alpha}(\mathcal{A}\llbracket A \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)}), \\
& \qquad c \mathbin{\hat{\times}} \tau^+(d) = \hat{\mathit{false}},\ c \mathbin{\hat{\times}} \tau^+(\eta^+) \mathrel{\tilde{\not\vdash}} \tau^-(\eta^-)\} \vee \\
& \bigvee\{\kappa([(\tau^+(c), \tau^-(\eta^-)) \rightarrowtail \tau^+(c)]^1 \cdot \alpha^\pm(r)\tilde{\downarrow}_{\tau^+(c)}) \mid \\
& \qquad \kappa([\mathit{stutt}(\tau^-(\eta^-))]^1 \cdot \alpha^\pm(r)) \in \tilde{\alpha}(\mathcal{A}\llbracket A \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)}) \\
& \qquad \tau^+(c) \mathrel{\tilde{\not\vdash}} \tau^-(\eta^-)\} \vee \\
& \{[(\hat{\mathit{true}}, \tau^-(\{c\})) \rightarrowtail \hat{\mathit{true}}]^1 \cdot \boxtimes \mid \boxtimes \in \tilde{\alpha}(\mathcal{A}\llbracket B \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)})\} \vee \\
& \bigvee\{ \kappa([(\tau^+(\eta^+), c \mathbin{\check{\times}} \tau^-(\eta^-)) \rightarrowtail \tau^+(d)]^1 \cdot \alpha^\pm(r)) \mid \\
& \qquad \kappa([(\tau^+(\eta^+), \tau^-(\eta^-)) \rightarrowtail \tau^+(d)]^1 \cdot \alpha^\pm(r)) \in \tilde{\alpha}(\mathcal{A}\llbracket B \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)}),\ \tau^+(\eta^+) \mathrel{\tilde{\not\vdash}} \tau^-(\{c\})\} \vee \\
& \bigvee\{\kappa([(\hat{\mathit{true}}, c \mathbin{\check{\times}} \tau^-(\eta^-)) \rightarrowtail \hat{\mathit{true}}]^1 \cdot \alpha^\pm(r)) \mid \kappa([\mathit{stutt}(\tau^-(\eta^-))]^1 \cdot \alpha^\pm(r)) \in \tilde{\alpha}(\mathcal{A}\llbracket B \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)})\} \\
& \quad [\,\text{by Definition of } \kappa \text{ (4.2.14)}\,] \\
= {} & \{[(\tau^+(c), \check{\mathit{false}}) \rightarrowtail \tau^+(c)]^1 \cdot \boxtimes \mid \boxtimes \in \tilde{\alpha}(\mathcal{A}\llbracket A \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)})\} \vee \\
& \bigvee\{ \kappa([(c \mathbin{\hat{\times}} \hat{\eta}, \check{\eta}) \rightarrowtail c \mathbin{\hat{\times}} \hat{d}]^n \cdot (\tilde{r}\tilde{\downarrow}_{\tau^+(c)})) \mid c \mathbin{\hat{\times}} \hat{d} \neq \hat{\mathit{false}}, \\
& \qquad c \mathbin{\hat{\times}} \hat{\eta} \mathrel{\tilde{\not\vdash}} \check{\eta},\ [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}]^{n+1} \cdot \tilde{r} \in \tilde{\alpha}(\mathcal{A}\llbracket A \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)})\} \vee \\
& \bigvee\{[(c \mathbin{\hat{\times}} \hat{\eta}, \check{\eta}) \rightarrowtail \hat{\mathit{false}}]^1 \mid [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}]^n \cdot \tilde{r} \in \tilde{\alpha}(\mathcal{A}\llbracket A \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)}), \\
& \qquad\qquad\qquad\qquad c \mathbin{\hat{\times}} \hat{d} = \hat{\mathit{false}},\ c \mathbin{\hat{\times}} \hat{\eta} \mathrel{\tilde{\not\vdash}} \check{\eta}\} \vee \\
& \bigvee\{\kappa([(\tau^+(c), \check{\eta}) \rightarrowtail \tau^+(c)]^1 \cdot [\mathit{stutt}(\check{\eta})]^n \cdot \tilde{r}\tilde{\downarrow}_{\tau^+(c)}) \mid [\mathit{stutt}(\check{\eta})]^{n+1} \cdot \tilde{r} \in \tilde{\alpha}(\mathcal{A}\llbracket A \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)}) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \tau^+(c) \mathrel{\tilde{\not\vdash}} \check{\eta}\} \vee \\
& \{[(\hat{\mathit{true}}, \tau^-(\{c\})) \rightarrowtail \hat{\mathit{true}}]^1 \cdot \boxtimes \mid \boxtimes \in \tilde{\alpha}(\mathcal{A}\llbracket B \rrbracket_{\gamma(\mathcal{I}^\alpha)})\} \vee \\
& \bigvee\{ \kappa([(\hat{\eta}, c \mathbin{\check{\times}} \check{\eta}) \rightarrowtail \hat{d}]^n \cdot \tilde{r}) \mid \\
& \qquad [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}]^n \cdot \tilde{r} \in \tilde{\alpha}(\mathcal{A}\llbracket B \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)}),\ \hat{\eta} \mathrel{\tilde{\not\vdash}} \tau^-(\{c\})\} \vee \\
& \bigvee\{\kappa([(\hat{\mathit{true}}, c \mathbin{\check{\times}} \check{\eta}) \rightarrowtail \hat{\mathit{true}}]^1 \cdot [\mathit{stutt}(\check{\eta})]^n \cdot \tilde{r}) \mid [\mathit{stutt}(\check{\eta})]^{n+1} \cdot \tilde{r} \in \tilde{\alpha}(\mathcal{A}\llbracket B \rrbracket_{\tilde{\gamma}(\mathcal{I}^\pm)})\} \\
& \quad [\,\text{by Inductive Hypothesis}\,]
\end{aligned}$$

$$\begin{aligned}
= {} & \{[(\tau^+(c), \check{\mathit{false}}) \rightarrowtail \tau^+(c)]^1 \cdot \boxtimes \mid \boxtimes \in \mathcal{A}^{\pm}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\} \vee \\
& \bigvee\{\, \kappa([(c \mathbin{\hat{\times}} \hat{\eta}, \check{\eta}) \rightarrowtail c \mathbin{\hat{\times}} \hat{d}]^n \cdot (\tilde{r}\tilde{\downarrow}_{\tau^+(c)})) \mid c \mathbin{\hat{\times}} \hat{d} \neq \widehat{\mathit{false}}, \\
& \qquad\qquad c \mathbin{\hat{\times}} \hat{\eta} \mathrel{\tilde{\nvDash}} \check{\eta},\ [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}]^{n+1} \cdot \tilde{r} \in \mathcal{A}^{\pm}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\} \vee \\
& \bigvee\{[(c \mathbin{\hat{\times}} \hat{\eta}, \check{\eta}) \rightarrowtail \widehat{\mathit{false}}]^1 \mid [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}]^n \cdot \tilde{r} \in \mathcal{A}^{\pm}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}, \\
& \qquad\qquad\qquad\qquad\qquad\qquad c \mathbin{\hat{\times}} \hat{d} = \widehat{\mathit{false}},\ c \mathbin{\hat{\times}} \hat{\eta} \mathrel{\tilde{\nvDash}} \check{\eta}\} \vee \\
& \bigvee\{\kappa([(\tau^+(c), \check{\eta}) \rightarrowtail \tau^+(c)]^1 \cdot [\mathit{stutt}(\check{\eta})]^n \cdot \tilde{r}\tilde{\downarrow}_{\tau^+(c)}) \mid [\mathit{stutt}(\check{\eta})]^{n+1} \cdot \tilde{r} \in \mathcal{A}^{\pm}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \tau^+(c) \mathrel{\tilde{\nvDash}} \check{\eta}\} \vee \\
& \{[(\widehat{\mathit{true}}, \tau^-(\{c\})) \rightarrowtail \widehat{\mathit{true}}]^1 \cdot \boxtimes \mid \boxtimes \in \mathcal{A}^{\pm}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\} \vee \\
& \bigvee\{\, \kappa([(\hat{\eta}, c \mathbin{\check{\times}} \check{\eta}) \rightarrowtail \hat{d}]^n \cdot \tilde{r}) \mid \\
& \qquad\qquad [(\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d}]^n \cdot \tilde{r} \in \mathcal{A}^{\pm}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})},\ \hat{\eta} \mathrel{\tilde{\nvDash}} \tau^-(\{c\})\} \vee \\
& \bigvee\{\kappa([(\widehat{\mathit{true}}, c \mathbin{\check{\times}} \check{\eta}) \rightarrowtail \widehat{\mathit{true}}]^1 \cdot [\mathit{stutt}(\check{\eta})]^n \cdot \tilde{r}) \mid [\mathit{stutt}(\check{\eta})]^{n+1} \cdot \tilde{r} \in \mathcal{A}^{\pm}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\} \\
= {} & \mathcal{A}^{\pm}[\![\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\mathcal{I}^{\pm}}
\end{aligned}$$

$\underline{\boldsymbol{A = A \parallel B}}\lrcorner$ This case is straightforward by Lemma 4.3.6.

$$\begin{aligned}
\tilde{\alpha}(\mathcal{A}[\![A \parallel B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}) &= \tilde{\alpha}(\bigsqcup\{r_A \mathbin{\bar{\|}} r_B \mid r_A \in \mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}, r_B \in \mathcal{A}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\}) \\
&= \bigvee\{\tilde{\alpha}(r_A \mathbin{\bar{\|}} r_B) \mid r_A \in \mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}, r_B \in \mathcal{A}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\} \\
&\qquad [\,\text{by Lemma 4.3.6}\,] \\
&= \bigvee\{\tilde{\alpha}(r_A) \mathbin{\tilde{\|}} \tilde{\alpha}(r_B) \mid r_A \in \mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}, r_B \in \mathcal{A}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}\} \\
&= \bigvee\{\tilde{r}_A \mathbin{\tilde{\|}} \tilde{r}_B \mid \tilde{r}_A \in \tilde{\alpha}(\mathcal{A}[\![A]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})}), \tilde{r}_B \in \tilde{\alpha}(\mathcal{A}[\![B]\!]_{\tilde{\gamma}(\mathcal{I}^{\pm})})\} \\
&\qquad [\,\text{by Inductive Hypothesis}\,] \\
&= \bigvee\{\tilde{r}_A \mathbin{\tilde{\|}} \tilde{r}_B \mid \tilde{r}_A \in \mathcal{A}^{\pm}[\![A]\!]_{\mathcal{I}^{\pm}}, \tilde{r}_B \in \mathcal{A}^{\pm}[\![B]\!]_{\mathcal{I}^{\pm}}\} \\
&\qquad [\,\text{by (4.3.5)}\,] \\
&= \mathcal{A}^{\pm}[\![A_1 \parallel A_2]\!]_{\mathcal{I}^{\pm}}
\end{aligned}$$

$\underline{\boldsymbol{A = \exists x\, A_1}}\lrcorner$ This case is straightforward by Lemma 4.3.8.

$$\begin{aligned}
\tilde{\alpha}(\mathcal{A}[\![\exists x\, A_1]\!]_{\tilde{\gamma}(\mathcal{I})}) &= \tilde{\alpha}(\bigsqcup\{\, \bar{\exists}_x\, r \mid r \in \mathcal{A}[\![A_1]\!]_{\tilde{\gamma}(\mathcal{I})}, r \text{ is } x\text{-self-sufficient}\}) \\
&= \bigvee\{\, \tilde{\alpha}(\bar{\exists}_x\, r) \mid r \in \mathcal{A}[\![A_1]\!]_{\tilde{\gamma}(\mathcal{I})}, r \text{ is } x\text{-self-sufficient}\} \\
&\qquad [\,\text{by Lemma 4.3.8}\,] \\
&= \bigvee\{\, \kappa(\tilde{\exists}_x\, \tilde{\alpha}(r)) \mid r \in \mathcal{A}[\![A_1]\!]_{\tilde{\gamma}(\mathcal{I})}, r \text{ is } x\text{-self-sufficient}\} \\
&\qquad [\,\text{by (4.3.1)}\,] \\
&= \bigvee\{\, \kappa(\tilde{\exists}_x\, \tilde{r}) \mid \tilde{r} \in \tilde{\alpha}(\mathcal{A}[\![A_1]\!]_{\tilde{\gamma}(\mathcal{I})}), \tilde{r} \text{ is abstractly } x\text{-self-sufficient}\} \\
&\qquad [\,\text{by Inductive Hypothesis}\,] \\
&= \bigvee\{\, \kappa(\tilde{\exists}_x\, \tilde{r}) \mid \tilde{r} \in \mathcal{A}^{\pm}[\![A_1]\!]_{\mathcal{I}^{\pm}}, \tilde{r} \text{ is abstractly } x\text{-self-sufficient}\} \\
&\qquad [\,\text{by (4.3.6)}\,] \\
&= \mathcal{A}^{\pm}[\![\exists x\, A_1]\!]_{\mathcal{I}^{\pm}}
\end{aligned}$$

$\underline{\boldsymbol{A = p(z)}}\lrcorner$

$$
\begin{aligned}
\tilde{\alpha}(\mathcal{A}[\![p(z)]\!]_{\tilde{\gamma}(\mathcal{I})}) &= \tilde{\alpha}(\bigsqcup\{(\mathit{true},\varnothing) \rightarrowtail \mathit{true}\cdot r \ \mid r \in \tilde{\gamma}(\mathcal{I})(p(z))\})\\
&= \bigvee\{\tilde{\alpha}((\mathit{true},\varnothing) \rightarrowtail \mathit{true}\cdot r) \ \mid r \in \tilde{\gamma}(\mathcal{I})(p(z))\}\\
&\qquad [\,\text{by (4.2.23)}\,]\\
&= \bigvee\{\kappa([(\tau^{+}(\mathit{true}),\tau^{-}(\varnothing)) \rightarrowtail \tau^{+}(\mathit{true})]^{1}\cdot\tilde{\alpha}(r)) \ \mid r \in \tilde{\gamma}(\mathcal{I})(p(z))\}\\
&\qquad [\,\text{since } \tau^{+}(\mathit{true}) = \widehat{\mathit{true}} \text{ and } \tau^{-}(\varnothing) = \check{\mathit{false}}\,]\\
&= \bigvee\{\kappa([(\widehat{\mathit{true}},\check{\mathit{false}}) \rightarrowtail \widehat{\mathit{true}}]^{1}\cdot\tilde{r}) \ \mid \tilde{r} \in \tilde{\alpha}(\tilde{\gamma}(\mathcal{I}^{\pm}(p(z))\}))\\
&\qquad [\,\tilde{\alpha}\circ\tilde{\gamma} \text{ is the identity for } \tilde{\mathbf{M}}^{\pm}\,]\\
&= \bigvee\{\kappa([(\widehat{\mathit{true}},\check{\mathit{false}}) \rightarrowtail \widehat{\mathit{true}}]^{1}\cdot\tilde{r}) \ \mid \tilde{r} \in \mathcal{I}^{\pm}(p(z))\}\\
&\qquad [\,\text{by (4.3.7)}\,]\\
&= \mathcal{A}^{\pm}[\![p(z)]\!]_{\mathcal{I}^{\pm}}
\end{aligned}
$$

5

Abstract Diagnosis for tccp based on temporal formulas

Abstract

Automatic techniques for program verification usually suffer the well-known state explosion problem. Most of the classical approaches are based on browsing the structure of some form of model (which represents the behavior of the program) to check if a given specification is valid. This implies that a subset of the model has to be built, and sometimes the needed fragment is quite huge.

In this chapter, we provide an alternative automatic decision method to check whether a given property, specified in a linear temporal logic, is *valid* w.r.t. a *tccp* program. Our proposal (based on abstract interpretation techniques) does not require to build any model at all.

More specifically, we provide an extension of the abstract diagnosis framework for *tccp* which works on an abstract domain formed by linear temporal formulas. This extension encompasses the limitations of the method presented in Chapter 4 where specifications were big and unnatural to be written. Given the impossibility of defining an abstraction function from the domain of conditional traces into the domain of temporal formulas, we introduce a "weaker" version of abstract diagnosis which is based only on the concretization function.

Our proposal intuitively consists in viewing a program P as a formula transformer and thus, in order to decide the validity of the specification S, we just have to check if the P-transformation of S implies S itself (i.e., if S is a pre-fixpoint of the transformation w.r.t. P). Thus, the final step of the abstract diagnosis process needs to check whether an implication formula is valid. We provide an automatic decision procedure for the temporal logic used, which, due to the underlying concurrent constraint model, has some differences w.r.t. the classical propositional LTL.

Modeling and verifying concurrent and reactive systems is a really complicated task which is crucial in a lot of modern applications. Thus, the development of automatic and efficient tools to formal verify these systems is essential. One of the most known techniques for formal verification of these systems is model checking. Model checking was originally introduced in [20, 100] to automatically check if a finite-state system satisfies a given property. It consisted in an exhaustive analysis of the state-space of the system; thus, the state-explosion problem is its main drawback and, for this reason, many proposals in the literature try to mitigate it. Some of the more successful ones are the symbolic approach [17, 68, 13], on-the-fly model checking [71] and the abstract interpretation based techniques [23, 40]. The idea which is shared by these approaches is to reduce the number of states of the system.

The model-checking technique for *tccp* was first defined in [53], and also in this setting (optimized) symbolic and abstract versions were later defined [3, 4].

In this chapter, we propose a completely different approach to the formal verification of LTL properties for concurrent and reactive systems specified in *tccp*. The linear temporal logic we use to express specifications, csLTL, is an adaptation of the propositional LTL logic to the concurrent constraint framework, following the ideas of [94, 44, 45, 116]. It is expressive enough to represent the abstract semantics of *tccp* with much precision. In brief, we formalize a method to validate a specification, expressed by an csLTL formula ϕ, of the expected behavior of a *tccp* program P which does not require to build any model at all. Namely, we define an extension of the abstract diagnosis for *tccp* defined in Chapter 4 where the abstract domain is formed by csLTL formulas. This proposal intuitively consists in viewing P as a formula transformer by means of an (abstract) immediate consequence operator $\dot{\mathcal{D}}[\![P]\!]$ which works on csLTL formulas. Then, to decide the validity of ϕ, we just have to check if $\dot{\mathcal{D}}[\![P]\!]_\phi$ (i.e., the P-transformation of ϕ) implies ϕ. We provide an automatic decision procedure for csLTL which, due to the underlying concurrent constraint model, has some differences w.r.t. the classical propositional LTL [58, 60].

The main motivation behind this proposal is to verify a LTL property for a *tccp* program P without building the model of P (which is usually quite big). This approach aims also to encompass the limitations of the instances presented in the formulation of Chapter 4 where specifications were big and onerous to be written.

The chapter is organized as follows. In Section 5.1 we provide a general scheme of abstract diagnosis for *tccp* which is defined parametrically w.r.t. a suitable family of concretization functions. This scheme can be used in case the abstraction function cannot be provided, as in the case of temporal formulas. In Section 5.2 the csLTL logic is introduced. In Section 5.3 we define an abstract semantics over csLTL formulas that correctly approximates the small-step behavior of *tccp* programs. This semantics is formally related to the concrete semantics of Section 3.1 by means of a concretization function. In Section 5.4 some examples of the application of abstract diagnosis over the domain of csLTL formulas are illustrated. Finally, in Section 5.5 a tableau construction algorithm for csLTL is presented in order to automatically check the validity of the abstract diagnosis test. All the proofs and the most technical definitions and results can be found in the chapter appendix 5.A.

5.1 Abstract Diagnosis for *tccp* based on concretization functions

Sometimes, defining a Galois Insertion between a concrete and an abstract domain **A** is not possible. For instance, this happens in case some concrete element has no best abstract approximation in **A**.

Consider, for example, the polyhedra approximation defined in [38]. The authors approximate a set of vector of reals $S \in \mathbf{R}^n$ with a polyhedron P such that $S \subseteq P$. If S is finite, the best correct approximation of P is the convex hull of S. However, a sphere has no best upper approximation by a convex polyhedron, since we obtain an infinite strictly decreasing chain of polyhedrons each one correctly approximating the sphere, but no one is the best abstract approximation.

Another example is the approximation of traces that represent the behavior of a programs by means of temporal logic formulas. In this case, we can have a formula ϕ that correctly approximate an infinite trace, but we can always find another formula ψ that is more precise then ϕ and is still a correct approximation of the trace. Therefore, as in the case of polyhedra approximation, no best abstract approximation is available.

In this situation, we cannot use the abstract diagnosis framework defined in Section 4.1 (which is actually parametric w.r.t. a Galois Insertion $\langle\alpha, \gamma\rangle$). Therefore, by using ideas from [35], we propose a new approach to abstract diagnosis which is defined only over the concretization function γ.

In order to guarantee the correctness of the method, we assume the concretization function γ to be *monotonic*, *injective* and $\sqcap$-*distributive*. Furthermore, $\mathbf{A}$ has to be a lattice (not necessarily complete) of the form $(\mathbf{A}, \leq, \bigvee, \bigwedge, \top, \bot)$.

The notions of correctness and completeness are defined similarly w.r.t. the general framework of Section 4.1.

Definition 5.1.1 *Given a set of declarations D and $\mathcal{S}^\alpha \in \mathbf{I_A}$, which is the specification of the abstract intended behavior of D over $\mathbf{A}$, we say that*

1. *D is (abstractly)* partially correct *w.r.t. $\mathcal{S}^\alpha$ if $\mathcal{F}[\![D]\!] \sqsubseteq \gamma(\mathcal{S}^\alpha)$.*
2. *D is (abstractly)* complete *w.r.t. $\mathcal{S}^\alpha$ if $\gamma(\mathcal{S}^\alpha) \sqsubseteq \mathcal{F}[\![D]\!]$.*

In this framework, *symptoms* are the differences between $\mathcal{F}[\![D]\!]$ and $\gamma(\mathcal{S}^\alpha)$ and the notions of *abstractly incorrect process declaration* and *uncovered element* are the same of Definition 4.1.2.

Now consider $\mathcal{D}^\alpha$ be a *monotonic* abstract immediate consequence operator which is a *correct approximation* of $\mathcal{D}$, i.e., given $D \in \mathbf{D}_\mathbf{C}^\Pi$ and $\mathcal{S}^\alpha \in \mathbf{I_A}$, $\mathcal{D}[\![D]\!]_\mathcal{S} \sqsubseteq \gamma(\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha})$.

Theorem 4.1.3 and Theorem 4.1.4 are reformulated as follows by using the new notions of partial correctness and completeness based on γ.

Theorem 5.1.2 *Let $D \in \mathbf{D}_\mathbf{C}^\Pi$ and $\mathcal{S}^\alpha \in \mathbf{I_A}$.*

1. *If there are no abstractly incorrect process declarations in D (i.e., $\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$), then D is partially correct w.r.t. $\mathcal{S}^\alpha$ (i.e., $\mathcal{F}[\![D]\!] \sqsubseteq \gamma(\mathcal{S}^\alpha)$).*
2. *Let D be partially correct w.r.t. $\mathcal{S}^\alpha$. If D has abstract uncovered elements, then D is not complete (i.e., $\gamma(\mathcal{S}^\alpha) \not\sqsubseteq \mathcal{F}[\![D]\!]$).*

Let us recall that the absence of abstractly incorrect declarations is a sufficient condition for partial correctness, but it is not necessary. Because of the approximation, it can happen that a (concretely) correct declaration is abstractly incorrect. Hence, abstract incorrect declarations are in general just a warning about a possible source of errors.

Similarly to the general approach of Section 4.1, an abstract correct declaration cannot contain an error; thus, no (manual) inspection is needed for declarations which are not abstractly incorrect. Moreover, as shown by the following theorem, all concrete errors—that are "visible"—are detected, as they lead to an abstract incorrectness or abstract uncovered. Intuitively, in this setting, a concrete error is *visible* if it is possible to express a formula ϕ whose concretization reveals the error (i.e., if the logic is expressible enough). This is stated formally in the following theorem.

Theorem 5.1.3 *Let R be a process declaration for $p(\vec{x})$, $\mathcal{S} \in \mathbf{I}$ a concrete specification and $\mathcal{S}^\alpha \in \mathbf{I_A}$ a sound approximation for $\mathcal{S}$ (i.e., $\mathcal{S} \sqsubseteq \gamma(\mathcal{S}^\alpha)$).*

1. *If $\mathcal{D}[\![\{R\}]\!]_{\mathcal{S}} \not\sqsubseteq \gamma(\mathcal{S}^\alpha)$ and it exists e such that $\gamma(e) \sqsubseteq \mathcal{D}[\![\{R\}]\!]_{\mathcal{S}}(p(\vec{x}))$ and $e \wedge \mathcal{S}^\alpha(p(\vec{x})) = \bot$, then R is abstractly incorrect w.r.t. $\mathcal{S}^\alpha$ (on testimony e).*

2. *If there exists an abstract uncovered element e w.r.t. $\mathcal{S}^\alpha$, then there exists $r \in \gamma(e)$ such that $r \notin \mathcal{D}[\![\{R\}]\!]_{\mathcal{S}}(p(\vec{x}))$.*

Point 1 can be read as: the concrete error has an abstract symptom which is not hidden by the approximation on $\mathcal{S}^\alpha$ and, moreover, there exists an abstract element e whose concretization reveals the error.

The main drawback of this setting w.r.t. the one presented in Section 4.1 is that, due to the lack of a Galois Insertion, the best correct approximation of $\mathcal{D}$ cannot be defined. However, in [38], several methods to obtain a correct approximation of $\mathcal{D}$ in the absence of a best correct approximation are listed.

In the following, we will show an example of lack of best abstract approximation for the concrete domain $\mathbf{M}$ and an instance of this scheme over an abstract domain of temporal formulas.

5.2 Abstraction scheme

Behavioral properties of *tccp* programs can be expressed in a natural way by using an appropriate temporal logic.

To this end, we define an abstract domain of logic formulas which is based on a variation of the classical Linear Temporal Logic [78]. Following [94, 44, 45, 116], the idea is to replace atomic propositions by constraints of the underlying constraint system.

Definition 5.2.1 (csLTL formulas) *Given a cylindric constraint system $\mathbf{C}$, $c \in \mathbf{C}$ and $x \in Var$, formulas of the* Constraint System Linear Temporal Logic *over $\mathbf{C}$ are defined by using the grammar:*

$$\phi ::= \dot{true} \mid \dot{false} \mid c \mid \dot{\neg}\,\phi \mid \phi \mathbin{\dot{\wedge}} \phi \mid \dot{\exists}_x\,\phi \mid \bigcirc\,\phi \mid \phi\,\mathcal{U}\,\phi.$$

We denote with **csLTL$_\mathbf{C}$** *the set of all temporal formulas over $\mathbf{C}$ (we omit $\mathbf{C}$ when clear from the context).*

The formulas $\dot{true}$, $\dot{false}$ and connectives $\dot{\neg}$, $\dot{\wedge}$ have the classical logical meaning. The atomic formula $c \in \mathbf{C}$ states that c has to be entailed by the current store. $\dot{\exists}_x\,\phi$ is the existential quantification over the set of variables Var. $\bigcirc\,\phi$ states that ϕ holds at the next time instant, while $\phi_1\,\mathcal{U}\,\phi_2$ states that ϕ_2 eventually holds and in all previous instants ϕ_1 holds. In the sequel (as usual), we use $\phi_1 \mathbin{\dot{\vee}} \phi_2$ as a shorthand for $\dot{\neg}\,\phi_1 \mathbin{\dot{\wedge}} \dot{\neg}\,\phi_2$; $\phi_1 \mathbin{\dot{\rightarrow}} \phi_2$ for $\dot{\neg}\,\phi_1 \mathbin{\dot{\vee}} \phi_2$; $\phi_1 \mathbin{\dot{\leftrightarrow}} \phi_2$ for $\phi_1 \mathbin{\dot{\rightarrow}} \phi_2 \mathbin{\dot{\wedge}} \phi_2 \mathbin{\dot{\rightarrow}} \phi_1$; $\diamond\,\phi$ for $\dot{true}\,\mathcal{U}\,\phi$ and $\Box\,\phi$ for $\dot{\neg}\,\diamond\,\dot{\neg}\,\phi$. $\diamond\,\phi$ holds if at some point in the future ϕ is true, and $\Box\,\phi$ holds if ϕ holds in the current instant and always in the future.

A *constraint formula* is an atomic formula c or its negation $\dot{\neg}\,c$. Formulas of the form $\bigcirc\,\phi$ and $\dot{\neg}\bigcirc\,\phi$ are called *next* formulas. Constraint and next formulas are said to be *elementary* formulas. Finally, formulas of the form $\phi_1\,\mathcal{U}\,\phi_2$ (or $\diamond\,\phi$ or $\dot{\neg}(\Box\,\phi)$) are called *eventualities.*

We define the abstract domain $\mathbf{F} := \mathsf{csLTL}/_{\dot\leftrightarrow}$ (i.e., the domain formed by csLTL formulas modulo logical equivalence) ordered by $\dot\rightarrow$. The algebraic structure $(\mathbf{F}, \dot\rightarrow, \dot\bigvee, \dot\bigwedge, \dot{true}, \dot{false})$ is a bounded join-semilattice, since:

1. it has a bottom element: $\dot{false} = \dot\bigwedge_{\phi\in\mathsf{csLTL}} \phi$;
2. the *lub* $\dot\vee$ of $\phi_1, \phi_2 \in \mathsf{csLTL}$, defined as $\phi_1 \dot\vee \phi_2$, exists and it is a temporal formula.

However, this structure is not a complete lattice, since both $\dot\bigwedge$ and $\dot\bigvee$ always exist just for finite sets of formulas (but not for infinite ones).

The semantics of a temporal formula is typically defined in terms of an infinite sequence of states which validates it. Here we use conditional traces instead.

As usually done in the context of temporal logics, we define the satisfaction relation $\vDash$ only for infinite conditional traces. We implicitly transform finite traces (which end in $\boxtimes$) by replicating the last store infinite times. Namely, the trace $(\eta_1^+, \eta_1^-) \rightarrowtail c_1 \dots (\eta_n^+, \eta_n^-) \rightarrowtail c_n \cdot \boxtimes$ becomes $(\eta_1^+, \eta_1^-) \rightarrowtail c_1 \dots (\eta_n^+, \eta_n^-) \rightarrowtail c_n \cdot (c_n, \varnothing) \rightarrowtail c_n \cdots (c_n, \varnothing) \rightarrowtail c_n \cdots$, while $\boxtimes$ becomes $(true, \varnothing) \rightarrowtail true \cdots (true, \varnothing) \rightarrowtail true \cdots$.

Definition 5.2.2 *The* semantics *of $\phi \in \mathbf{F}$ is given by $\gamma^{\mathbf{F}}\colon \mathbf{F} \to \mathbf{M}$ defined as*

$$\gamma^{\mathbf{F}}(\phi) := \bigsqcup\{r \in \mathbf{M} \mid r \vDash \phi\}, \tag{5.2.1}$$

where, for each $\phi, \phi_1, \phi_2 \in$ csLTL, $c, d, \eta^+ \in \mathbf{C}$, $\eta^- \subseteq \mathbf{C}$ and $r, r' \in \mathbf{M}$, the satisfaction relation $\vDash$ is defined as:

$$r \vDash \dot{true} \tag{5.2.2a}$$

$$r \nvDash \dot{false} \tag{5.2.2b}$$

$$(\eta^+, \eta^-) \rightarrowtail d \cdot r' \vDash c \qquad \text{iff } \eta^+ \vdash c \tag{5.2.2c}$$

$$stutt(\eta^-) \cdot r' \vDash c \qquad \text{iff } \forall d^- \in \eta^-. c \nvdash d^- \text{ and } r' \vDash c \tag{5.2.2d}$$

$$r \vDash \dot\neg \phi \qquad \text{iff } r \nvDash \phi \tag{5.2.2e}$$

$$r \vDash \phi_1 \dot\wedge \phi_2 \qquad \text{iff } r \vDash \phi_1 \text{ and } r \vDash \phi_2 \tag{5.2.2f}$$

$$r \vDash \dot\exists_x \phi \qquad \text{iff it exists } r' \text{ such that } \bar\exists_x r' = \bar\exists_x r,\ r' \text{ } x\text{-self-sufficient and } r' \vDash \phi \tag{5.2.2g}$$

$$r \vDash \bigcirc \phi \qquad \text{iff } r^1 \vDash \phi \tag{5.2.2h}$$

$$r \vDash \phi_1 \,\mathcal{U}\, \phi_2 \qquad \text{iff } \exists i \geq 1. \forall j < i.\ r^i \vDash \phi_2 \text{ and } r^j \vDash \phi_1 \tag{5.2.2i}$$

We say that $\phi \in \mathbf{F}$ is a sound approximation *of $R \in \mathbf{M}$ if $R \sqsubseteq \gamma^{\mathbf{F}}(\phi)$.*

By abusing in notation, we extend the notion of $\vDash$ to sets of formulas in the following way

$$r \vDash \Phi \iff \forall \phi \in \Phi.\ r \vDash \phi \tag{5.2.3}$$

A formula ϕ is said to be satisfiable *if there exists $r \in \mathbf{M}$ such that $r \vDash \phi$, while it is said to be* valid *if, for all $r \in \mathbf{M}$, $r \vDash \phi$.*

All the cases are fairly standard except (5.2.2c) and (5.2.2d). The conditional trace $r = (\eta^+, \eta^-) \rightarrowtail d \cdot r'$ prescribes that η^+ is entailed by the current store, thus r models all the constraint formulas c such that $\eta^+ \vdash c$. We have to note that, by the monotonicity of the store during *tccp* computations, the positive conditions in conditional traces contain all the information previously added in the constraint store.

Furthermore, by the definition of condition, since η^+ cannot be in contradiction with η^-, it holds that neither c is in contradiction with η^-. Thus, the conditional trace $\mathit{stutt}(\eta^-)\cdot r'$ models all the constraint formulas c that are not in contradiction with the set η^- and such that, by monotonicity, c holds in the continuation r'.

Lemma 5.2.3 *The function $\gamma^{\mathbf{F}}$ is monotonic, injective and $\bigsqcap$-distributive.*

In order to use the classical abstract interpretation approach for Galois Insertions $\langle\alpha,\gamma\rangle$, we have to determine if $\gamma^{\mathbf{F}}$ admits its adjoint abstraction function. This is equivalent to the existence, for each $R \in \mathbf{M}$, of $\dot{\bigwedge}\{\phi \in \mathbf{F} \mid R \sqsubseteq \gamma^{\mathbf{F}}(\phi)\}$. However, this is false. For instance, given $R := \{(\mathit{true},\varnothing) \rightarrowtail x > 0 \cdot (x > 0,\varnothing) \rightarrowtail x > 1 \ldots (x > n,\varnothing) \rightarrowtail x > n+1 \ldots\}$, the set $\{\phi \,|\, R \sqsubseteq \gamma^{\mathbf{F}}(\phi)\}$ has no *glb* (in $\mathbf{F}$). This is equivalent to say that no best abstract approximation is available for R in $\mathbf{F}$, as explained in Section 5.1.

Thus, in the sequel, we cannot use the classical abstract interpretation approach for abstract diagnosis based on Galois Insertions $\langle\alpha,\gamma\rangle$ (see Section 4.1). Therefore, we use the above proposed weaker version of abstract diagnosis for join-semilattices which is based only on the concretization function γ (see Section 5.1).

In the following, we instantiate this framework with the csLTL domain. We first define an appropriate and sound semantics and, then, we redefine in this setting the notions of abstractly incorrect declaration and uncovered element.

5.3 csLTL Abstract Semantics

In this section we define an abstract semantics for *tccp* and we show that is a correct approximation of the concrete semantics of Section 3.1. This semantics associates to each *tccp* program a csLTL formula which approximate its small-step behavior.

The technical core of this semantics definition is the csLTL agent semantics evaluation function $\dot{\mathcal{A}}[\![A]\!]$ which, given an agent A and an interpretation $\dot{\mathcal{I}}$ (for the process symbols of A), builds a csLTL formula which is a sound approximation of the (concrete) behavior of A. In the sequel, we denote by $\mathbf{A}^{\Pi}_{\mathbf{C}}$ the set of agents and $\mathbf{D}^{\Pi}_{\mathbf{C}}$ the set of sets of process declarations built on signature Π and constraint system $\mathbf{C}$.

Analogously to Definition 5.3.1, we define interpretations over the domain $\mathbf{F}$ as functions $\mathbf{PC} \to \mathbf{F}$ modulo variance.

Definition 5.3.1 *Let* $\mathbf{PC} := \{p(\vec{x}) \mid p \in \Pi,\ \vec{x}$ *are distinct variables*$\}$. *An* interpretation *is a function* $\mathbf{PC} \to \mathbf{F}$ *modulo variance. Two functions* $I, J\colon \mathbf{PC} \to \mathbf{F}$ *are* variants *if for each* $\pi \in \mathbf{PC}$ *there exists a renaming* ρ *such that* $(I\pi)\rho = J(\pi\rho)$. *The semantic domain* $\mathbf{I}_{\mathbf{F}}$ *is the set of all interpretations ordered by the point-wise extension of* $\dot{\to}$.

Definition 5.3.2 (csLTL Semantics) *Given* $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$ *and* $\dot{\mathcal{I}} \in \mathbf{I}_{\mathbf{F}}$, *we define the* csLTL semantics evaluation $\dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}}$ *by structural induction as follows.*

$$\dot{\mathcal{A}}[\![\mathsf{skip}]\!]_{\dot{\mathcal{I}}} := \dot{\mathit{true}} \tag{5.3.1a}$$

$$\dot{\mathcal{A}}[\![\mathsf{tell}(c)]\!]_{\dot{\mathcal{I}}} := \bigcirc\, c \tag{5.3.1b}$$

$$\dot{\mathcal{A}}[\![\textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\dot{\mathcal{I}}} := \Box(\dot{\bigwedge}_{i=1}^{n} \dot{\neg}\, c_i) \,\dot{\vee}\, \Big((\dot{\bigwedge}_{i=1}^{n} \dot{\neg}\, c_i)\, \mathcal{U} \,\dot{\bigvee}_{i=1}^{n} \big(c_i \,\dot{\wedge} \bigcirc \dot{\mathcal{A}}[\![A_i]\!]_{\dot{\mathcal{I}}}\big)\Big) \tag{5.3.1c}$$

$$\dot{\mathcal{A}}[\![\mathsf{now}\ c\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2]\!]_{\dot{\mathcal{I}}} := (c \mathbin{\dot\wedge} \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}) \mathbin{\dot\vee} (\dot\neg c \mathbin{\dot\wedge} \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}) \tag{5.3.1d}$$

$$\dot{\mathcal{A}}[\![A_1 \parallel A_2]\!]_{\dot{\mathcal{I}}} := \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}} \mathbin{\dot\wedge} \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}} \tag{5.3.1e}$$

$$\dot{\mathcal{A}}[\![\exists x\, A]\!]_{\dot{\mathcal{I}}} := \dot\exists_x\, \dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}} \tag{5.3.1f}$$

$$\dot{\mathcal{A}}[\![p(\vec{x})]\!]_{\dot{\mathcal{I}}} := \bigcirc \dot{\mathcal{I}}(p(\vec{x})) \tag{5.3.1g}$$

Given $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$ *we define the immediate consequence operator* $\dot{\mathcal{D}}[\![D]\!] : \mathbf{I_F} \to \mathbf{I_F}$ *as*

$$\dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{I}}}(p(\vec{x})) := \dot{\bigvee} \left\{ \dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}} \mid p(\vec{x}) :- A \in D \right\}$$

It can be noticed that $\dot{\mathcal{A}}$ and $\dot{\mathcal{D}}$ are monotonic, as stated formally by the following lemma.

Lemma 5.3.3 *For each* $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$ *and each* $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$*,* $\dot{\mathcal{A}}[\![A]\!]$ *and* $\dot{\mathcal{D}}[\![D]\!]$ *are monotonic.*

Let us recall from Chapter 2 a (non trivial) example of *tccp* program that we use through this chapter to illustrate the achievements of our proposal.

Example 5.3.4

The following two process declarations model in *tccp* a part of a railway crossing system (the full declarations can be found in [6] or in Chapter 2 of this thesis).

```
$controller(C, G) :- \exists C', G' \Big($
        now $(C = [near \mid \_])$ then
            tell$(C = [near \mid C'])$ ∥ tell$(G = [down \mid G'])$ ∥ $controller(C', G')$
        else  now $(C = [out \mid \_])$ then
                tell$(C = [out \mid C'])$ ∥ tell$(G = [up \mid G'])$ ∥ $controller(C', G')$
              else $controller(C, G)\Big)$
```

The controller process uses an *input channel* C (implemented as a stream) through which it receives signals from the environment (trains), and an *output channel* G through which it sends orders to the gate process. It checks the input channel for a *near* signal (the guard in the first now agent), in which case it sends (tells) the order *down* through G, links the future values (C') of the stream C and restarts the check at the following time instant (recursive call *controller*(C', G')). If the *near* signal is not detected, then, the else branch looks for the *out* signal and (if present) behaves dually to the first branch. Finally, if no signal is detected at the current time instant (last else branch), then the process keeps checking from the following time instant (the process call takes one time instant). The gate process reacts to the signals from the controller:

```
$gate(G, S) :- \exists G', S' \Big($
  ask$(G = [down \mid \_]) \to$
    $\big($tell$(G = [down \mid G'])$ ∥ ask$(true)^{100} \to ($tell$(S = [down \mid S'])$ ∥ $gate(G', S')))\big)$
  + ask$(G = [up \mid \_]) \to$
    $\big($tell$(G = [up \mid G'])$ ∥ ask$(true)^{100} \to ($tell$(S = [up \mid S'])$ ∥ $gate(G', S')))\big)\Big)$
```

where $\mathsf{ask}(true)^n$ denotes the n-times repetition of the agent $\mathsf{ask}(true)$, and it corresponds to a delay of n time units. Through the input channel G, orders are received, and the state of the gate (represented by the stream S) is consequently updated. The ask agent (with two branches) makes the gate wait (suspend) until one of the guards is entailed, i.e., until one of the two orders is received. Once a signal is detected, after 100 time instants, the state of the gate is appropriately updated and a recursive call is done in order to keep the gate active (i.e., waiting for the successive order).

Now, we show an example of calculus of the semantics $\dot{\mathcal{D}}$ for the railway crossing system.

Example 5.3.5

Consider the set of declarations D_{rc} of Example 5.3.4 and let us use $\bigcirc^n$ to abbreviate the repetition of $\bigcirc$ n-times. Given $\dot{\mathcal{I}} \in \mathbf{I_F}$, with Definition 5.3.2 we compute

$$\begin{aligned}
&\dot{\mathcal{D}}[\![D_{rc}]\!]_{\dot{\mathcal{I}}}(controller(C,G))) = \phi_M(\dot{\mathcal{I}}) \coloneqq \phi_{near}(\dot{\mathcal{I}}) \mathbin{\dot\vee} \phi_{out}(\dot{\mathcal{I}}) \mathbin{\dot\vee} \phi_{cwait}(\dot{\mathcal{I}})\\
&\dot{\mathcal{D}}[\![D_{rc}]\!]_{\dot{\mathcal{I}}}(gate(G,S)) = \phi_g(\dot{\mathcal{I}}) \coloneqq \big(\phi_{gwait}\,\mathcal{U}\,(\phi_{down}(\dot{\mathcal{I}}) \mathbin{\dot\vee} \phi_{up}(\dot{\mathcal{I}}))\big) \mathbin{\dot\vee} \Box\,\phi_{gwait}
\end{aligned}$$

where

$$\begin{aligned}
\phi_{near}(\dot{\mathcal{I}}) &= \dot\exists_{C',G'}\,\big(C = [near \mid \_] \mathbin{\dot\wedge} \bigcirc C = [near \mid C'] \mathbin{\dot\wedge}\\
&\qquad \bigcirc G = [down \mid G'] \mathbin{\dot\wedge} \bigcirc\dot{\mathcal{I}}(controller(C',G'))\big)\\
\phi_{out}(\dot{\mathcal{I}}) &= \dot\exists_{C',G'}\,\big(\dot\neg(C = [near \mid \_]) \mathbin{\dot\wedge} \bigcirc C = [out \mid C'] \mathbin{\dot\wedge}\\
&\qquad C = [out \mid \_] \mathbin{\dot\wedge} \bigcirc G = [up \mid G'] \mathbin{\dot\wedge} \bigcirc\dot{\mathcal{I}}(controller(C',G'))\big)\\
\phi_{cwait}(\dot{\mathcal{I}}) &= \dot\neg(C = [near \mid \_]) \mathbin{\dot\wedge} \dot\neg(C = [out \mid \_]) \mathbin{\dot\wedge} \bigcirc\dot{\mathcal{I}}(controller(C,G))\\
\phi_{gwait} &= \dot\neg(G = [down \mid \_]) \mathbin{\dot\wedge} \dot\neg(G = [up \mid \_])\\
\phi_{down}(\dot{\mathcal{I}}) &= \dot\exists_{G',S'}\,\big(G = [down \mid \_] \mathbin{\dot\wedge} \bigcirc\big(\bigcirc G = [down \mid S'] \mathbin{\dot\wedge}\\
&\qquad \bigcirc^{100}(\bigcirc S = [down \mid S'] \mathbin{\dot\wedge} \bigcirc\dot{\mathcal{I}}(gate(G',S')))\big)\big)\\
\phi_{up}(\dot{\mathcal{I}}) &= \dot\exists_{G',S'}\,\big(G = [up \mid \_] \mathbin{\dot\wedge} \bigcirc\big(\bigcirc G = [up \mid G'] \mathbin{\dot\wedge}\\
&\qquad \bigcirc^{100}(\bigcirc S = [up \mid S'] \mathbin{\dot\wedge} \bigcirc\dot{\mathcal{I}}(gate(G',S')))\big)\big)
\end{aligned}$$

The three disjoints of $\phi_M(\dot{\mathcal{I}})$ match the three possible behaviors of $controller(C,G)$: when signal *near* is emitted by the train ($\phi_{near}(\dot{\mathcal{I}})$), when *out* is emitted ($\phi_{out}(\dot{\mathcal{I}})$), and when no signal arrives ($\phi_{cwait}(\dot{\mathcal{I}})$). Similarly, the formula $\phi_g(\dot{\mathcal{I}})$ states that, either the process waits forever, or when a signal is received, then it changes the state of the gate ($\phi_{down}(\dot{\mathcal{I}})$ and $\phi_{up}(\dot{\mathcal{I}})$).

As stated by the following theorem, $\dot{\mathcal{A}}$ is a sound approximation of $\mathcal{A}$ and $\dot{\mathcal{D}}$ is a sound approximation of $\mathcal{D}$.

Theorem 5.3.6 (Correctness of $\dot{\mathcal{A}}$ and $\dot{\mathcal{D}}$) *Let $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$, $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$ and $\dot{\mathcal{I}} \in \mathbf{I_F}$. Then, $\mathcal{A}[\![A]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})} \sqsubseteq \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}})$ and $\mathcal{D}[\![D]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})} \sqsubseteq \gamma^{\mathbf{F}}(\dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{I}}})$.*

5.4 Abstract Diagnosis for *tccp* based on csLTL formulas

As already claimed, given the impossibility of defining a Galois Insertion between $\mathbf{M}$ and $\mathbf{F}$, we cannot use the abstract diagnosis framework for *tccp* defined in Section 4.1 which is actually parametric w.r.t. a Galois insertion $\langle\alpha,\gamma\rangle$. Instead, we use the weaker version defined in Section 5.1 which is parametric to a given concretization functions.

Let us reformulate the notions of abstractly incorrect process declaration and uncovered element in the csLTL context.

Definition 5.4.1 *Let $D \in \mathbf{D}_{\mathbf{C}}^{\Pi}$, R a process declaration for process p, $\phi_t \in \mathbf{F}$ and $\dot{\mathcal{S}} \in \mathbf{I_F}$.*

- *R is* abstractly incorrect *w.r.t. $\dot{\mathcal{S}}$ (on testimony ϕ_t) if $\phi_t \dot{\to} \dot{\mathcal{D}}[\![\{R\}]\!]_{\dot{\mathcal{S}}}(p(\vec{x}))$ and $\phi_t \dot{\wedge} \dot{\mathcal{S}}(p(\vec{x})) = \dot{false}$, or equivalently if $\dot{\mathcal{D}}[\![\{R\}]\!]_{\dot{\mathcal{S}}}(p(\vec{x})) \not\dot{\to} \dot{\mathcal{S}}(p(\vec{x}))$.*
- *ϕ_t is an* uncovered element *for $p(\vec{x})$ w.r.t. $\dot{\mathcal{S}}$ if $\phi_t \dot{\to} \dot{\mathcal{S}}(p(\vec{x}))$ and $\phi_t \dot{\wedge} \dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{S}}}(p(\vec{x})) = \dot{false}$.*

From Theorem 5.1.2 and Theorem 5.1.3 we can summarize the main results of abstract diagnosis over csLTL formulas as follows:

- If $\dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{S}}} \dot{\to} \dot{\mathcal{S}}$ then D is partially correct w.r.t. $\dot{\mathcal{S}}$ (i.e., $\mathcal{F}[\![D]\!] \sqsubseteq \gamma(\dot{\mathcal{S}})$).
- Let D be partially correct w.r.t. $\dot{\mathcal{S}}$. If D has abstract uncovered elements then D is not complete (i.e., $\gamma(\dot{\mathcal{S}}) \not\sqsubseteq \mathcal{F}[\![D]\!]$).
- All concrete errors—that are "visible"— are detected, as they lead to an abstract incorrectness or abstract uncovered. A concrete error is *visible* if it is possible to express a formula ϕ whose concretization reveals the error (i.e., if the logic is expressible enough).

We show some examples of this abstract diagnosis approach by using the abstract domain $\mathbf{F}$ and the concretization function $\gamma^{\mathbf{F}}$ (5.2.1). As usual, in the following examples, we borrow from [6] the notation for *last entailed value* of a stream: $X\dot{=}c$ holds if the last instantiated value in the stream X is c.

Example 5.4.2

Assume we want to check (for the railroad crossing system in Example 5.3.4) whether it holds that each time a *near* signal arrives from a train, the gate eventually is down. To model this property, we define the specification $\dot{\mathcal{S}}_{down}$ as:

$$\dot{\mathcal{S}}_{down}(controller(C,G)) \coloneqq \phi_{ordersent} \coloneqq \square(C\dot{=}near \dot{\to} \diamond(G\dot{=}down))$$
$$\dot{\mathcal{S}}_{down}(gate(G,S)) \coloneqq \phi_{gatedown} \coloneqq \square(G\dot{=}down \dot{\to} \diamond(S\dot{=}down))$$

To check whether $\dot{\mathcal{D}}[\![D_{rc}]\!]_{\dot{\mathcal{S}}_{down}} \dot{\to} \dot{\mathcal{S}}_{down}$ (see Example 5.3.5) we have to check if $\phi_M(\dot{\mathcal{S}}_{down}) \dot{\to} \phi_{ordersent}$ and $\phi_g(\dot{\mathcal{S}}_{down}) \dot{\to} \phi_{gatedown}$. Recall the fixpoint characterization of the temporal operators, i.e., $\square p = p \dot{\wedge} \bigcirc \square p$ and $\diamond p = p \dot{\vee} \bigcirc \diamond p$. It can be seen that each of the three disjoints of $\phi_M(\dot{\mathcal{S}}_{down})$ implies $\phi_{ordersent}$. For $\phi_g(\dot{\mathcal{S}}_{down})$, while the process is waiting, the antecedent of both implications cannot be derived, thus the formula is true. Moreover, when the order down arrives (in the second component of the until), it also occurs that the state is updated (see ϕ_{down} in Example 5.3.5). Thus $\phi_g(\dot{\mathcal{S}}_{down}) \dot{\to} \dot{\mathcal{S}}_{down}$ and then $\dot{\mathcal{D}}[\![D_{rc}]\!]_{\dot{\mathcal{S}}_{down}} \dot{\to} \dot{\mathcal{S}}_{down}$. Hence, by Theorem 4.1.3, D_{rc} is partially correct w.r.t. $\dot{\mathcal{S}}_{down}$.

Example 5.4.3

In this example we show how our technique detects an error in a buggy set of declarations obtained from D_{rc} by removing the instruction $\mathsf{tell}(G = [up \mid G'])$ in the definition of process *controller*. To avoid misunderstandings, we call the modified process *controller'* and let R be the modified process definition.

We want to check whether the order *up* is sent whenever the signal *out* is received. Thus, we define the specification:

$$\phi := \dot{\mathcal{S}}_{up}(controller'(C,G)) := \Box((C \dot{=} out) \dot{\to} \Diamond(G \dot{=} up))$$

We have

$$\phi' := \dot{\mathcal{D}}[\![\{R\}]\!]_{\dot{\mathcal{S}}_{up}}(controller'(C,G)) = \phi'_{near} \dot{\vee} \phi'_{out} \dot{\vee} \phi'_{cwait}$$

where

$$\begin{aligned}
\phi'_{near} &= \dot{\exists}_{C',G'} \big(C = [near \mid \_] \dot{\wedge} \bigcirc C = [near \mid C'] \dot{\wedge} \\
&\qquad \bigcirc G = [down \mid G'] \dot{\wedge} \bigcirc \dot{\mathcal{S}}_{up}(controller'(C',G'))\big) \\
\phi'_{out} &= \dot{\exists}_{C',G'} \big(\dot{\neg}(C = [near \mid \_]) \dot{\wedge} C = [out \mid \_] \dot{\wedge} \\
&\qquad \bigcirc(C = [out \mid C'] \dot{\wedge} \bigcirc \dot{\mathcal{S}}_{up}(controller'(C',G')))\big) \\
\phi'_{cwait} &= \dot{\neg}(C = [near \mid \_]) \dot{\wedge} \dot{\neg}(C = [out \mid \_]) \dot{\wedge} \bigcirc \dot{\mathcal{S}}_{up}(controller'(C,G))
\end{aligned}$$

We detect an incorrectness of R w.r.t. $\dot{\mathcal{S}}_{up}$ on testimony $\phi_t := \phi'_{out} \dot{\wedge} \Box G \dot{=} down$ since $\phi_t \dot{\to} \phi'$ but $\phi_t \dot{\wedge} \phi = \mathit{false}$.

Example 5.4.4

Let us now check a "wrong" property for the gate process. We define the specification:

$$\phi_{updown} := \dot{\mathcal{S}}_{wrong}(gate(G,S)) := \Diamond(G \dot{=} up \dot{\vee} G \dot{=} down)$$

which states that eventually in the future either the order *up* or *down* is sent by the gate. We have

$$\begin{aligned}
\phi_g(\dot{\mathcal{S}}_{wrong}) :=& \dot{\mathcal{D}}[\![D_{rc}]\!]_{\dot{\mathcal{S}}_{wrong}}(gate(G,S)) \\
=& (\phi_{gwait}\, \mathcal{U}\, (\phi_{down}(\dot{\mathcal{S}}_{wrong}) \dot{\vee} \phi_{up}(\dot{\mathcal{S}}_{wrong}))) \dot{\vee} \Box \phi_{gwait}
\end{aligned}$$

It can be noticed that $\phi_g(\dot{\mathcal{S}}_{wrong}) \not\dot{\to} \phi_{updown}$, since $\Box \phi_{gwait} = \Box(\dot{\neg}(G \dot{=} up) \dot{\wedge} G \dot{=} down)$ does not imply $\Diamond(G \dot{=} up \dot{\vee} G \dot{=} down)$.

This is a warning about a possible error in the definition of gate process w.r.t. the specification $\dot{\mathcal{S}}_{wrong}$.

Example 5.4.5 (Pathological cases)

Let $D_p := \{q(y) \coloneq \mathsf{now}\ y = 1\ \mathsf{then}\ q(y)\ \mathsf{else}\ q(y)\}$. It is worth noticing that this program is a loop that does nothing at all since, independently from the check if $x = 1$, it calls itself.

Consider the specification $\dot{\mathcal{S}}_p(q(y)) := \Diamond(y = 1)$, we have

$$\dot{\mathcal{D}}[\![D_p]\!]_{\dot{\mathcal{S}}_p}(q(y)) = (x > 0 \dot{\wedge} \Diamond x = 1) \dot{\vee} (\dot{\neg} x > 0 \dot{\wedge} \Diamond x = 1), \text{ thus } \dot{\mathcal{D}}[\![D_p]\!]_{\dot{\mathcal{S}}_p} \dot{\to} \Diamond(y = 1)$$

and, by Theorem 4.1.3, D_p is partially correct w.r.t. $\dot{\mathcal{S}}_p$. However, it can be noticed that $y = 1$ is not explicitly added by the process $q(y)$.

Example 5.4.6

Let $D_x := \{p(x) \mathrel{:-} \mathsf{now}\ x = 1\ \mathsf{then\ skip\ else}\ (\mathsf{tell}(x = 1) \parallel p(x))\}$. For specification $\dot{\mathcal{S}}_x(p(x)) := \Diamond(x = 1)$, which states that eventually in the future $x = 1$ is entailed by the constraint store, we have that

$$\dot{\mathcal{D}}[\![D_x]\!]_{\dot{\mathcal{S}}_x}(p(x)) = (x = 1) \mathbin{\dot{\vee}} (\dot{\neg}\, x = 1 \mathbin{\dot{\wedge}} \bigcirc x = 1 \mathbin{\dot{\wedge}} \bigcirc \Diamond(x = 1)).$$

Thus $\dot{\mathcal{D}}[\![D_x]\!]_{\dot{\mathcal{S}}_x} \mathrel{\dot{\rightarrow}} \Diamond(x = 1)$ and, by Theorem 4.1.3, D_x is partially correct w.r.t. $\dot{\mathcal{S}}_x$. In fact, if $x = 1$ is entailed by the current constraint store, the program stops, otherwise the else branch is taken and $x = 1$ is added in the constraint store by the tell agent, in both cases $x = 1$ is eventually entailed.

Example 5.4.5 shows a negative phenomenon of our methodology, which in general happens for sets of declarations D where $\dot{\mathcal{D}}[\![D]\!]$ has more than one fixpoint (this essentially happens when D contains a loop which does not produce contributes, not for meaningful programs). In such situation we can have that the actual behavior does not model a specification $\dot{\mathcal{S}}$ which is a non-least fixpoint of $\dot{\mathcal{D}}[\![D]\!]$, but we do not detect abstractly incorrect declarations since $\dot{\mathcal{S}}$ is a fixpoint. However, if $\dot{\mathcal{S}}(p(\vec{x}))$ is assumed to hold for each process $p(\vec{x})$ defined in D and $\dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{S}}} \mathrel{\dot{\rightarrow}} \dot{\mathcal{S}}$, then $\mathcal{F}[\![D]\!]$ satisfies $\dot{\mathcal{S}}$.

To conclude this section, we would like to point out that with our method we have validated/unvalidated all the properties of systems already present in the *tccp* literature.

5.5 An automatic decision procedure for csLTL

In order to make our abstract diagnosis approach effective, we need to define an automatic decision procedure to check the validity of the csLTL formulas that show up when checking a property. In particular, we need to handle csLTL formulas of the form $\psi \mathrel{\dot{\rightarrow}} \phi$, where ψ corresponds to the computed approximated behavior of the program, and ϕ is the abstract intended behavior of the process.

We impose a restriction on the specification ϕ: we do not allow the use of existential quantifications. Actually, this restriction is quite natural in our context since, in general, we are interested in proving properties related to the *visible* behavior of the program, not to the local variables. In contrast, negation can be applied to any formula ϕ (not only to constraints).

In this section, we extend the tableau construction for Propositional LTL (PLTL) of [60, 57] in order to deal with csLTL formulas. We need to adapt the method to our context due to three issues:

1. The structures on which the logic is interpreted are different. In our case, traces (sequences of states) are monotonic, meaning that the information in each state always increases.

2. The logic itself is a bit different from PLTL since propositions are replaced by constraints in **C**.

3. We have to handle existential quantification over variables of the underlying constraint system. This does not mean that we are dealing with a first-order logic as will become clear later.

	α	$A(\alpha)$
R1	$\dot\neg\dot\neg\phi$	$\{\phi\}$
R2	$\phi_1 \dot\wedge \phi_2$	$\{\phi_1, \phi_2\}$
R3	$\dot\neg\bigcirc\phi$	$\{\bigcirc\dot\neg\phi\}$

	β	$B_1(\beta)$	$B_2(\beta)$
R4	$\dot\neg(\phi_1 \dot\wedge \phi_2)$	$\{\dot\neg\phi_1\}$	$\{\dot\neg\phi_2\}$
R5	$\dot\neg(\phi_1\,\mathcal{U}\,\phi_2)$	$\{\dot\neg\phi_1, \dot\neg\phi_2\}$	$\{\phi_1, \dot\neg\phi_2, \dot\neg\bigcirc(\phi_1\,\mathcal{U}\,\phi_2)\}$
R6	$\phi_1\,\mathcal{U}\,\phi_2$	$\{\phi_2\}$	$\{\phi_1, \dot\neg\phi_2, \bigcirc((\Gamma^* \dot\wedge \phi_1)\,\mathcal{U}\,\phi_2)\}$

Figure 5.1: α- and β-formulas rules

In the following, we first present the basic rules that are used during the construction of the tree associated to the tableau. Then we present the algorithm that implements the process of construction of the tree.

5.5.1 Basic rules for a csLTL tableau

Classic tableaux algorithms are based on the systematic construction of a graph which is used to check the satisfiability of the formula. In [58] the authors present a first algorithm that does not need auxiliary structures (such as graphs) to decide about the satisfaction of the formula. This makes this approach more suitable for automatization. In [60, 57] this algorithm was slightly modified and improved in order to gain more efficiency.

A tableau procedure is defined by means of rules that build a tree whose nodes are labeled with sets of formulas. If all branches of the tree are *closed*, then the formula has no models. Otherwise, we can obtain a model that satisfies the formula from an open branch. Let us introduce the basic rules for the csLTL case. As usual, we present just the minimal set of rules.

A tableau rule is applied to a node n labeled with the set of formulas $L(n)$. Each rule application requires a previous selection of a formula ϕ from $L(n)$. We call context the set of formulas $L(n) \smallsetminus \{\phi\}$ and we denote it with Γ. Conjunctions are α-formulas and disjunctions β-formulas. Figure 5.1 presents the rules for α– and β–formulas.

Tables in Figure 5.1 are interpreted as follows. Each row in a table represents a rule. Each time that an α–rule is applied to a node of the tree, a formula of the node matching the pattern in column α is replaced in a child node by the corresponding $A(\alpha)$. For the β-rules, two children nodes are generated, one for each column $B_1(\beta)$ and $B_2(\beta)$.

Almost all the rules are standard. However, Rule R6 uses the so-called context Γ^*, which is defined as $\Gamma^* := \dot\bigvee_{\gamma\in\Gamma} \dot\neg\gamma$. The use of contexts is the mechanism to detect the loops where no formula changes, thus allowing to mark branches containing eventually formulas as *open*. This kind of rules were first used in [59]. The idea is that, by using contexts, loops where no formula changes are *discarded* since they cannot close a branch.

Note that there is no rule defined for the $\bigcirc$ operator. In fact, the $\mathsf{next}(\Phi)$ function transforms a set of elementary formulas Φ into another set: $\mathsf{next}(\Phi) := \{\phi \mid \bigcirc\phi \in \Phi\} \cup \{\dot\neg\phi \mid \dot\neg\bigcirc\phi \in \Phi\} \cup \{c \mid c \in \Phi, c \in \mathbf{C}\}$. This operator is different from the corresponding one of PLTL in that, in addition to keep the internal formula of the next formulas, it also *passes* the constraints that are entailed at the current time instant to the following one. This makes sense for *tccp* computations since, as already mentioned, the store in a computation is monotonic, thus no information can be removed and it happens that c always implies $\bigcirc c$.

The next operator is a key notion in the kind of tableaux defined in [58, 60, 57]. This operator allows one to identify *stages* in a tableau which represent time instants in the model.

We show that α- and β-formulas rules and the next operator preserve the satisfiability of a set of formulas.

Lemma 5.5.1 *Given a set of formulas Φ, an α-formula α and a β-formula β:*

1. $\Phi \cup \{\alpha\}$ *is satisfiable* $\Leftrightarrow$ $\Phi \cup A(\alpha)$ *is satisfiable;*
2. $\Phi \cup \{\beta\}$ *is satisfiable* $\Leftrightarrow$ $\Phi \cup B_1(\beta)$ *or* $\Phi \cup B_2(\beta)$ *is satisfiable;*
3. *if* Φ *is a set of elementary formulas,* Φ *is satisfiable* $\Leftrightarrow$ $\mathsf{next}(\Phi)$ *is satisfiable;*

A second main difference w.r.t. the PLTL case regards the existential quantification. The csLTL existential quantification does not correspond to the first-order logic one. It is introduced to model information about local variables, thus, the formula $\dot{\exists}_x \phi$ can be seen as the formula ϕ where the information about x is local.

We define a specific rule for the $\dot{\exists}$ case: when the selected formula of a given node is of the form $\dot{\exists}_x \phi$, it is created a node, child of n, whose labeling is that of n except that the formula $\dot{\exists}_x \phi$ is replaced by $\phi[y/x]$ with y fresh variable. Correctness of this rule derives from the following lemma, which shows that $\dot{\exists}_x \phi$ and ϕ are equi-satisfiable.

Lemma 5.5.2 *Let $\phi \in$ csLTL, $\dot{\exists}_x \phi$ is satisfiable $\iff$ ϕ is satisfiable.*

Corollary 5.5.3 *Let $\Phi \subseteq$ csLTL such that $y \in Var$ does not appear in Φ (y is a fresh variable) and let $\phi \in$ csLTL. Then, $\Phi \cup \{\dot{\exists}_x \phi\}$ is satisfiable $\iff$ $\Phi \cup \{\phi[y/x]\}$ is satisfiable.*

Proof.

Follows directly from Lemma 5.5.2. x does not appear in Φ, thus the local variable x of ϕ is independent from any other variable in Φ.

Intuitively, the renaming of a local variable x with a fresh variable is performed in order to avoid a clash between x and another variable with the same name that might appear in the tableau.

This approach works in our context since the operator $\dot{\exists}_x$ does not correspond to the existential quantifier of classical first-order LTL. Instead, it models the fact that the variable x is local to the process of interest. In fact, Lemma 5.5.2 and Corollary 5.5.3 do not hold in the first-order LTL in general, as shown by the following counterexample.

Example 5.5.4

Consider the first-order LTL formula $\phi := \Box(\dot{\exists}_x(x = 26 \mathbin{\dot{\wedge}} \bigcirc x \neq 26))$ where $\dot{\exists}$ is interpreted as the classical first order existential quantifier. It can be noticed that ϕ is satisfiable (in the classical fist order LTL interpretation with flexible variables) since it exists a sequence of stores that is a model of ϕ, i.e., $(x = 26) \cdot (x \neq 26 \wedge y = 26) \cdot (y \neq 26 \wedge z = 26) \cdots$.

Consider now the formula obtained by eliminating the existential quantifier in ϕ and by renaming x with a fresh name: $\phi' := \square(x' = 26 \dot{\wedge} \bigcirc x' \neq 26)$. It can be noticed that ϕ' is not satisfiable since it requires both $x' = 26$ and $x' \neq 26$ to hold from the second time instant on. Thus, in classical first order LTL ϕ is satisfiable but ϕ' not.

Let us now interpret ϕ and ϕ' in our csLTL framework. In *tccp*, the store is monotonic, thus variables are not flexible and cannot change their values as time passes. In this case both formulas are satisfiable: consider the conditional traces $r := (x = 26, \varnothing) \rightarrowtail \mathit{false} \cdot (\mathit{false}, \varnothing) \rightarrowtail \mathit{false} \cdot \boxtimes$ and $r' := (x' = 26, \varnothing) \rightarrowtail \mathit{false} \cdot (\mathit{false}, \varnothing) \rightarrowtail \mathit{false} \cdot \boxtimes$. From Definition 5.2.2 it follows that $r \vDash \phi$ and $r' \vDash \phi'$. Therefore, in csLTL both ϕ and ϕ' are satisfiable.

5.5.2 Semantic csLTL tableau

In this section, we present the notion of tableau for our csLTL formulas following the ideas of [60, 57]. For sake of clarity, since we borrow some definitions and notions from that work, in this section we skip some formal definitions (which can be find in the chapter appendix).

A tableau $\mathcal{T}_\Phi$ for a set of formulas Φ is a tree-like structure where each node n is labeled with a set of csLTL formulas $L(n)$. The root is labeled with the set of formulas Φ whose satisfiability/unsatisfiability is needed to check; Then, children of nodes are the result of applying the basic rules of Subsection 5.5.1. The algorithm in which these nodes are built is given in the following subsection. Nodes with no children are called *leaf* nodes.

Definition 5.5.5 (csLTL tableau) *A csLTL tableau for a finite set of formulas Φ is a tuple $\mathcal{T}_\Phi = (Nodes, n_\Phi, L, B, R)$ such that:*

1. *Nodes is a finite non-empty set of nodes;*
2. *$n_\Phi \in Nodes$ is the initial node;*
3. *$L : Nodes \to \wp(csLTL)$ is the labeling function that associates to each node the formulas which are true in that node; the initial node is labeled with Φ;*
4. *B is the set of branches such that exactly one of the following points holds for every $b = n_0, \ldots, n_i, n_{i+1}, \ldots, n_k \in B$ and every $0 \leq i < k$:*
   (a) *for an α-formula $\alpha \in L(n_i)$, $L(n_{i+1}) = \{A(\alpha)\} \cup L(n_i) \smallsetminus \{\alpha\}$;*
   (b) *for a β-formula $\beta \in L(n_i)$, $L(n_{i+1}) = \{B_1(\beta)\} \cup L(n_i) \smallsetminus \{\beta\}$ and there exists another branch in B of the form $b' = n_0, \ldots, n_i, n'_{i+1}, \ldots, n'_k$ such that $L(n'_{i+1}) = \{B_2(\beta)\} \cup L(n_i) \smallsetminus \{\beta\}$;*
   (c) *for an existential quantified formula $\dot{\exists}_x \phi' \in L(n_i)$, $L(n_{i+1}) = \{\phi''\} \cup L(n_i) \smallsetminus \{\dot{\exists}_x \phi'\}$ where $\phi'' := \phi'[y/x]$ with y fresh variable;*
   (d) *in case $L(n_i)$ is a set formed only by elementary formulas, $L(n_{i+1}) = \mathsf{next}(L(n_i))$, where $\mathsf{next}(\Phi) := \{\phi \mid \bigcirc \phi \in \Phi\} \cup \{\dot{\neg} \phi \mid \dot{\neg} \bigcirc \phi \in \Phi\} \cup (\Phi \cap \mathbf{C})$.*

A branch $b \in B$ is said to be maximal *if it is not a proper prefix of another branch in B.*

Rules 4a and 4b are standard, replacing α and β-formulas with one or two formulas according to the matching pattern of rules in Figure 5.1, except for Rule R6 that uses the so-called context Γ^*, which is defined in the following. The next operator used in Rule 4d is different from the corresponding one of PLTL since it also preserves the constraint formulas. This is needed for guaranteeing correctness since, as already mentioned, in *tccp* computations the store is monotonic, thus ($c \dot{\rightarrow} \bigcirc c$ and) constraint information has to be permanent.

Finally, Rule 4c is specific for the $\dot{\exists}$ case. $\dot{\exists}_x$ is removed after renaming x with a fresh variable.

Definition 5.5.6 *A node in the tableau is* inconsistent *if it contains*

- *a couple of formulas $\phi, \dot{\neg}\,\phi$, or*
- *the formula $\dot{false}$, or*
- *a constraint formula $\dot{\neg}\, c'$ such that the merge c of all the (positive) constraint formulas $c_1, \dots, c_n$ in the node ($c := c_1 \otimes \cdots \otimes c_n$) is such that $c \vdash c'$.*

The last condition for inconsistence of a node is particular to the *ccp* context. Since we are dealing with constraints that model partial information, it is possible to have an *implicit* inconsistence, in the sense that we need the entailment relation to detect it.

An inconsistent node does not accept any rule application. When a branch contains an inconsistent node, it is said to be closed, otherwise it is open.

By Lemma 5.5.1 and by Definition 5.5.5, it can be noticed that every closed branch contains only unsatisfiable sets of formulas. Open branches are not necessarily satisfiable since they could be prefixes of a closed one.

Similarly to the PLTL case, there exists only a finite number of different labels in a tableau (as stated formally by Proposition 5.A.11). Thus, if there exists an infinite branch $b = n_0, n_1, \dots n_k \dots$, it necessarily contains a cycle (i.e., contains infinitely many repetition of nodes with the same label). These branches are called *cyclic branches* and can be finitely represented as $\mathsf{path}(b) = n_0, n_1, \dots, n_j, (n_{j+1}, \dots, n_k)^\omega$ when $L(n_k) = L(n_j)$ for $0 \le j < k$. Every branch of a tableau is divided into stages, denoted by $\mathsf{stages}(b)$. A *stage* is a sequence of consecutive nodes between two consecutive applications of the next operator. We abuse of notation and denote by $L(s)$ the labeling of a stage s defined as $\bigcup_{n \in s} L(n)$. It can be noticed that if b contains a cyclic sequence of nodes, then $\mathsf{stages}(b)$ is a cyclic sequence of stages.

We borrow from [58, 60, 57] the characterization of *fulfilled* eventually formula in a path of the tableau, namely when it is satisfied. We say that, when an eventually formula $\phi_1 \,\mathcal{U}\, \phi_2$ (or $\diamond \phi_2$) belongs to the labeling of a stage s in a path, it is fulfilled if there exists a subsequent stage s' such that $\phi_2 \in L(n')$. A sequence of stages S is fulfilling if all the eventually formulas in its labeling are fulfilled in S and a branch b is fulfilling if the sequences of stages in its paths are fulfilling.

Finally, an open branch is expanded if it is fulfilling and all its stages are *saturated.*

Definition 5.5.7 ([60, 57]) *A stage s is* saturated *if no α-, β- or hiding rule can be applied to any of its nodes.*

An eventuality formula $\phi_1 \,\mathcal{U}\, \phi_2$ (or $\diamond \phi_2$) that belongs to the labeling of a stage s in a branch is fulfilled *if there exists a subsequent stage s' such that $\phi_2 \in L(n')$.*

A sequence of stages S is fulfilling *if all the eventuality formulas are fulfilled in S and a branch b is* fulfilling *if all* $\mathsf{stages}(b)$ *are fulfilling.*

An open branch is expanded *if it is fulfilling and all its stages are saturated.*

A tableau is called expanded *if every maximal branch is either expanded or closed.*

An expanded tableau is closed *if every branch ends in an inconsistent node, otherwise it is* open.

These notions are needed to formalize the tableau construction since only branches that are non-expanded and open are selected to be further developed.

Definition 5.5.8 (expanded csLTL tableau [60, 57]) *A tableau is called expanded if every branch is expanded or closed. An expanded tableau is closed if every branch ends in an inconsistent node, otherwise it is open.*

5.5.3 A systematic csLTL tableaux construction

We can define an algorithm to automatically build an expanded csLTL tableau (called systematic tableau) for a given set of formulas Φ along the lines of the one in [60, 57]. The construction consists in selecting at each step a non-expanded branch that can be extended by using α or β rules or $\dot{\exists}$ elimination. When none of these can be applied, the next operator is used to pass to the next stage. When dealing with eventualities, to determine the context Γ^* in Rule R6, it is necessary to *distinguish* the eventuality that is being unfolded in the path. Given a node n and $\phi \in L(n)$, $\Gamma := L(n) \setminus \{\phi\}$. Then, when Rule R6 is applied to a *distinguished* eventuality, we set $\Gamma^* := \dot{\bigvee}_{\gamma \in \Gamma} \dot{\neg}\, \gamma$; otherwise $\Gamma^* := \mathit{true}$. If a node does not contain any distinguished eventuality, then the algorithm distinguishes one of them and rule R6 is applied to it. Each node of the tableau has at most one distinguished eventuality.

The algorithm marks nodes when they cannot be further processed. In particular, a node is marked as *closed* when it is inconsistent and is marked as *open* when it contains just constraint formulas or when it is the last node of an expanded branch (all the eventualities in the branch are fulfilled).

Definition 5.5.9 (Systematic Tableau Algorithm) *Given a finite set of formulas Φ, the* systematic tableau $\mathcal{T}_\Phi$ *is built by repeatedly selecting an unmarked leaf node l and applying, in order, one of the points shown below.*

1. *Select an eventuality in l (if there is at least one) and distinguish it.*
2. *If l is an inconsistent node, then mark it as closed ($\times$).*
3. *If $L(l)$ is a set of constraint formulas, mark l as open ($\odot$).*
4. *Choose $\phi \in L(l)$ such that ϕ is not an* elementary *formula and it is not the distinguished eventuality. Then,*
   (a) *if ϕ is an existential quantified formula $\dot{\exists}_x\, \phi'$, then create a new node l' as a child of l and label it as $L(l') = (L(l) \setminus \{\phi\}) \cup \{\phi'\}$, where $\phi' := \phi[y/x]$ with y fresh variable;*
   (b) *if ϕ is an α-formula, create a new node l' as a child of l and label it as $L(l') = (L(l) \setminus \{\phi\}) \cup A(\phi)$ by using the corresponding rule in Figure 5.1;*

   (c) *if ϕ is a β-formula, create two new nodes l' and l'' as children of l and label them respectively as $L(l') = (L(l) \smallsetminus \{\phi\}) \cup B_1(\phi)$ and $L(l'') = (L(l) \smallsetminus \{\phi\}) \cup B_2(\phi)$ by using the corresponding rule in Figure 5.1. For Rule R6, when ϕ is an eventuality, we choose $\Gamma^* := true$.*

5. *When all the non distinguish formulas have been selected, apply Rule R6 with $\Gamma^* := \dot{\bigvee}_{\gamma\in\Gamma} \dot{\neg}\gamma$ to the distinguish eventuality ϕ: create two new nodes l' and l'' respectively as children of l and label them as $L(l') = (L(l) \smallsetminus \{\phi\}) \cup B_1(\phi)$ and $L(l'') = (L(l) \smallsetminus \{\phi\}) \cup B_2(\phi)$. Then, distinguish the* next *formula in $B_2(\beta)$;.*

6. *If $L(l)$ is a set of elementary formulas, then*

   (a) *if $L(l) = L(l')$ for l' ancestor of l (i.e., we detect a cycle), take the oldest ancestor of l that is labeled as $L(l)$ (denote it by l'') and check if all the eventualities in the path between l'' and l have been distinguished in such path. In this case mark l as open ($\odot$). Otherwise, apply the* next *operator: create a new node l''' as child of l and label it as $L(l''') = \mathsf{next}(L(l))$. Then, distinguish a new eventuality in $L(l''')$ following a fair strategy.*

   (b) *If no cycle has been detected, apply the* next *operator: create a new node l' as child of l and label it as $L(l') = \mathsf{next}(L(l))$. If ϕ is the distinguished formula in l and $\mathsf{next}(\phi) = \phi'$, ϕ' becomes the distinguished eventuality in l'. Otherwise, distinguish a new eventuality in $L(l')$ following a fair strategy.*

The construction terminates when every branch is marked.

By construction, each stage in the systematic tableau $\mathcal{T}_\Phi$ for Φ is saturated. In order to ensure the termination of the algorithm it is necessary to use a *fair* strategy to distinguish eventualities, in the sense that every eventuality in an open branch must be distinguished at some point. This assumption and the fact that, given a finite set of initial formulas, there exist only a finite set of possible labels in a systematic tableau, imply termination (as stated formally by Lemma 5.5.10).

It is worth noticing that, by the application of the rules in Figure 5.1, when both ϕ and $\neg\phi$ belong to the labeling of a stage in a branch b, then any branch prefixed by b is closed. Moreover, by construction, non-fulfilled undistinguished eventualities in a branch are maintained until they are fulfilled or they become distinguished.

One key result of the tableau in [60, 57] that we preserve is that if a distinguished eventuality is not fulfilled in an expanded branch b, then we can mark the branch as closed. This is because if we apply Rule R6, then we get a contradiction with the context of the eventuality.

Hence, we have that every distinguished eventuality in a cyclic branch b of $\mathcal{T}_\Phi$ is fulfilled, because otherwise b would be closed and then not cyclic. Also, by construction and the above properties, b is open if and only if the last node of b contains only constraint formulas, or b is cyclic and all its eventualities are fulfilled in b.

Lemma 5.5.10 *The algorithm of Definition 5.5.9 when using a fair strategy for the selection of eventualities, given as input a finite set $\Phi \subseteq \mathsf{csLTL}$, terminates and builds an expanded tableau for $\mathcal{T}_\Phi$.*

5.5.4 Soundness and completeness

Let us now show that the proposed algorithm is sound and complete for proving the satisfiability/unsatisfiability of csLTL formulas.

Theorem 5.5.11 (Soundness) *If there exists a closed systematic tableau for $\Phi \subseteq$ csLTL, then Φ is unsatisfiable.*

In order to prove completeness, we need to define an auxiliary function stores that, given a sequence of stages of the tableau, builds a suitable conditional trace which joins all the accumulated information in a stage at each time instant. We abuse of notation and write ϵ the empty sequence of stages. Recall that $\otimes$ is the join operation of the constraint system and $\bigotimes \varnothing = true$.

$$\begin{aligned}
&\mathsf{stores}(\epsilon) = \epsilon \\
&\mathsf{stores}(s \cdot S) = (d, \varnothing) \rightarrowtail d \cdot \mathsf{stores}(S) \qquad \text{where } d = \bigotimes\{c \mid c \in L(n) \cap \mathbf{C}, n \in s\}
\end{aligned}$$

By definition of next, which in our case propagates the constraints from one stage to the following, the conditional trace r resulting of applying stores to a sequence of stages S is monotone. Furthermore, since all the negative conditions are empty, r is also consistent.

We show that, given a systematic tableau $\mathcal{T}_\Phi$ built for Φ, we can compute a model for Φ from every open branch b in $\mathcal{T}_\Phi$.

Lemma 5.5.12 *Let b be an* open expanded *branch in the systematic tableau $\mathcal{T}_\Phi$ for $\Phi \subseteq$ csLTL. Given the sequence of stages S in* path(b)*, then* stores$(S) \models \Phi$.

Theorem 5.5.13 (Completeness) *If $\Phi \subseteq$ csLTL is satisfiable, then there exists a finite open tableau for Φ.*

5.5.5 Application of the tableau

The systematic tableau algorithm of Definition 5.5.9 can be used to check the implication resulting from the application of abstract diagnosis to the domain of csLTL formulas (Section 5.4). Thanks to Theorem 5.5.11, to check the validity of a formula of the form $\psi \dot{\rightarrow} \phi$, with $\phi = \dot{\mathcal{S}}(p(\vec{x}))$ and $\psi = \dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{S}}}(p(\vec{x}))$, we just have to build the tableau for its negation $\mathcal{T}_{\dot{\neg}(\psi \dot{\rightarrow} \phi)}$ and check if it is closed or not. If it is, we have that D is abstractly correct.

Otherwise, by the following Proposition 5.5.14, we have that from $\mathcal{T}_{\dot{\neg}(\psi \dot{\rightarrow} \phi)}$ we can extract an explicit testimony φ of the abstract incorrectness of D, since $\varphi \dot{\rightarrow} \dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{S}}}(p(\vec{x}))$ and $\varphi \dot{\nrightarrow} \dot{\mathcal{S}}(p(\vec{x}))$.

Proposition 5.5.14 *Let $\mathcal{T}_\Phi$ be an open systematic tableau for $\Phi = \{\psi, \dot{\neg}\phi\}$, b be an open branch in $\mathcal{T}_\Phi$, φ_i be the conjunction of the constraint formulas occurring in the i-th stage of b and φ be $\varphi_1 \dot{\wedge} \bigcirc \varphi_2 \ldots \dot{\wedge} \bigcirc^n \varphi_n$. Then $\varphi \dot{\rightarrow} \psi$ and $\varphi \dot{\nrightarrow} \phi$.*

The construction of $\psi = \dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{S}}}(p(\vec{x}))$ is linear in the size of D. The systematic tableau construction of $\dot{\neg}(\psi \dot{\rightarrow} \phi)$ (from what said in [60]) has worst case $O(2^{O(2^{|\dot{\neg}(\psi \dot{\rightarrow} \phi)|})})$. However, the worst-case asymptotic behavior in this context is quite meaningless since it is not very realistic to think that the formulas of the specification should grow much (big formulas are difficult to comprehend and in real situations people would hardly try even to imagine them). Consequently, we would not have big implications $\psi \dot{\rightarrow} \phi$, since ψ is bounded by ϕ. Moreover, note that tableau explosion is due to nesting of eventualities and in practice we have really few of them. Therefore, in real situations, we do not expect that (extremely) big tableaux will be built.

Let us show two examples of construction of the systematic tableaux for two formulas of this kind.

Example 5.5.15

Let us assume that we are trying to check whether process

$$R := p(y) :- \exists x\,(\mathsf{now}\ y = 1\ \mathsf{then}\ \mathsf{tell}(x = 5) \parallel p(y)\ \mathsf{else}\ \mathsf{tell}(y = 1))$$

satisfies $\dot{\mathcal{S}}(p(\vec{x})) := \Diamond(y = 1)$. Since $\dot{\mathcal{D}}[\![\{R\}]\!]_{\dot{\mathcal{S}}} = \dot{\exists}_x\,\phi$, where

$$\phi = (y = 1 \mathbin{\dot\wedge} \bigcirc x = 5 \mathbin{\dot\wedge} \bigcirc(\Diamond y = 1)) \mathbin{\dot\vee} (\dot\neg y = 1 \mathbin{\dot\wedge} \bigcirc y = 1)$$

Thus, we have to check if $\dot{\exists}_x\,\phi \mathbin{\dot{\rightarrow}} \Diamond(y = 1)$. Figure 5.2 shows the systematic tableau built for the negation of the formula, i.e., $\dot{\exists}_x\,\phi \mathbin{\dot\wedge} \Box\dot\neg(y = 1)$. Arrows labeled with α and

$$\begin{tikzcd}
 & \underline{\dot\exists_x\,\phi \mathbin{\dot\wedge} \Box(\dot\neg y = 1)} \arrow[d, "\alpha"] & \\
 & \underline{\dot\exists_x\,\phi}, \Box(\dot\neg y = 1) \arrow[d, "\exists"] & \\
 & \underline{\phi'}, \Box(\dot\neg y = 1) \arrow[dl, "\beta"] \arrow[dr] & \\
\underline{y = 1 \mathbin{\dot\wedge} \bigcirc x' = 5 \mathbin{\dot\wedge} \bigcirc(\Diamond y = 1)}, \Box(\dot\neg y = 1) \arrow[d, "\alpha"] & & \underline{\dot\neg y = 1 \mathbin{\dot\wedge} \bigcirc y = 1}, \Box(\dot\neg y = 1) \arrow[d, "\alpha"] \\
y = 1, \bigcirc x' = 5, \bigcirc(\Diamond y = 1), \underline{\Box(\dot\neg y = 1)} \arrow[d, "\alpha"] & & \dot\neg y = 1, \bigcirc y = 1, \underline{\Box(\dot\neg y = 1)} \arrow[d, "\alpha"] \\
\begin{array}{c} y = 1, \bigcirc x' = 5, \bigcirc(\Diamond y = 1), \\ \dot\neg y = 1, \bigcirc\Box(\dot\neg y = 1) \\ \times \end{array} & & \dot\neg y = 1, \bigcirc y = 1, \bigcirc(\Box(\dot\neg y = 1)) \arrow[d, "X"] \\
 & & y = 1, \underline{\Box(\dot\neg y = 1)} \arrow[d, "\alpha"] \\
 & & \begin{array}{c} y = 1, \dot\neg y = 1, \bigcirc(\Box(\dot\neg y = 1)) \\ \times \end{array}
\end{tikzcd}$$

Figure 5.2: Tableau for $\dot\exists_x\,\phi \mathbin{\dot{\rightarrow}} \Diamond y = 1$ of Example 5.5.15.

β correspond to the application of α and β rules, respectively; arrows labeled with X represent the application of the next operator. Finally, arrows labeled with $\exists$ correspond to the elimination of the existential quantification.

In the example, the first step uses the rule for the conjunction. Then, the second step involves the elimination of the existential quantification for $\dot{\exists}_x\,\phi$, with ϕ' we indicate the renaming of ϕ where x is replaced by the fresh variable x'.

The formula ϕ' is then selected for a β step (disjunction). The branch on the left is closed after two steps since $y = 1$ and $\dot{\neg}\, y = 1$ both belong to the node labeling.

The branch on the right, first flattens the conjunction and then applies the next rule. Note that the negation of a constraint is not kept in the following time instant. We recall that negation means "not entailment" (in contrast to meaning that *the contrary is true*), thus, in the future, the constraint could become true.

Since both branches are closed, we know that the formula $\dot{\exists}_x\,\phi \mathbin{\dot{\wedge}} \square\,\dot{\neg}(y = 1)$ is not satisfiable, thus its negation $\dot{\exists}_x\,\phi \mathbin{\dot{\rightarrow}} \lozenge(y = 1)$ is valid.

In the context of abstract diagnosis, this proves that the program is abstractly correct w.r.t. the csLTL specification.

Example 5.5.16

Let us consider a program with a single process declaration $D := \{p(y) \mathrel{:-} A\}$, where

$$A := \exists x\,(\mathsf{now}\ y = 1\ \mathsf{then}\ (\mathsf{tell}(x = 5) \parallel p(y))\ \mathsf{else}\ \mathsf{tell}(y = 1))$$

Now, suppose that we want to check that the constraint $y = 1$ is always entailed by the store. The corresponding specification is $\dot{\mathcal{S}}(p(y)) = \square(y = 1)$.

The csLTL-semantics $\dot{\mathcal{D}}$ for $p(y)$ using the given specification as interpretation is $\dot{\exists}_x\big((y = 1 \mathbin{\dot{\wedge}} \bigcirc x = 5 \mathbin{\dot{\wedge}} \bigcirc(\square y = 1)) \mathbin{\dot{\vee}} (\dot{\neg}\, y = 1 \mathbin{\dot{\wedge}} \bigcirc y = 1)\big)$. Let us abbreviate the body of the existential quantification as ϕ. To check whether the process $p(y)$ is correct w.r.t. the property, we have to show that $\dot{\exists}_x\,\phi \mathbin{\dot{\rightarrow}} \square(y = 1)$ is valid.

Figure 5.3 shows part of the (finite) tableau that proves the satisfiability of the formula $\dot{\exists}_x\,\phi \mathbin{\dot{\wedge}} \lozenge(\dot{\neg}\, y = 1)$. This means that its negation, $\dot{\exists}_x\,\phi \mathbin{\dot{\rightarrow}} \square(y = 1)$, is not valid. In the context of abstract diagnosis, although the formula is actually not satisfied by the program, because of the loss of precision due to the approximation, this is only a warning about the possible incorrectness of the program w.r.t. the csLTL specification.

Notice that the second step involves the elimination of the existential quantification $\dot{\exists}_x$: we denote with ϕ' the renaming of ϕ where x is replaced by the fresh variable x'. Furthermore, Rule R6 is applied twice to deal with the distinguished eventuality $\lozenge(\dot{\neg}\, y = 1)$.

5.6 Related Work

A Constraint Linear Temporal Logic is defined in [116] for the verification of *ntcc*, which shares with *tccp* the concurrent constraint nature and the non-monotonic behavior. A fragment of the proposed logic, the restricted negation fragment where negation is only allowed for state formulas, is shown to be decidable. However, no efficient decision procedure is given (apart from the proof itself). Moreover, the verification results are given for the locally-independent fragment of *ntcc*, which avoids the non-monotonicity of the original language.

A model-checking technique for *tccp* was formalized in [53], where the constraint nature of the language was exploited as a means to mitigate the state-explosion problem. In [3, 4], two optimizations employing symbolic representations and abstract interpretation

$$\begin{tikzcd}[column sep=tiny]
 & & & \underline{\dot\exists_x \phi \mathbin{\dot\wedge} \Diamond(\dot\neg\, y = 1)} \arrow[d, "\alpha"] & & & \\
 & & & \underline{\dot\exists_x \phi}, \Diamond(\dot\neg\, y = 1) \arrow[d, "\exists"] & & & \\
 & & & \underline{\phi'}, \Diamond(\dot\neg\, y = 1) \arrow[dll, "\beta"'] \arrow[drr, "\beta"] & & & \\
 & \underline{y = 1 \mathbin{\dot\wedge} \bigcirc x' = 5 \mathbin{\dot\wedge} \bigcirc(\Box\, y = 1)}, \Diamond(\dot\neg\, y = 1) \arrow[d, "\alpha"] & & & & \underline{\dot\neg\, y = 1 \mathbin{\dot\wedge} \bigcirc y = 1}, \Diamond(\dot\neg\, y = 1) \arrow[d, "\alpha"] & \\
 & y = 1, \bigcirc x' = 5, \bigcirc(\Box\, y = 1), \underline{\Diamond(\dot\neg\, y = 1)} \arrow[dl, "\beta"] \arrow[dddr] & & & & \dot\neg\, y = 1, \bigcirc y = 1, \underline{\Diamond(\dot\neg\, y = 1)} \arrow[dl, "\beta"] \arrow[dddr] & \\
 y = 1, \bigcirc x' = 5, \bigcirc(\Box\, y = 1), \dot\neg\, y = 1 & & & & \dot\neg\, y = 1, \bigcirc y = 1 \arrow[d, "X"] & & \\
 \times & & & & y = 1 \quad \odot & & \\
 & & y = 1, \bigcirc(\Box\, y = 1), \bigcirc x' = 5, (\dot\neg\, y = 1 \mathbin{\dot\vee} \dot\neg \bigcirc(\Box\, y = 1))\, \mathcal{U}\, (\dot\neg\, y = 1) \arrow[dl, dash, dashed] \arrow[dr, dash, dashed] & & & & \dot\neg\, y = 1, \bigcirc y = 1, \bigcirc(y = 1 \mathbin{\dot\vee} \dot\neg \bigcirc y = 1)\, \mathcal{U} (\dot\neg\, y = 1) \arrow[dl, dash, dashed] \arrow[dr, dash, dashed] \\
 & {} & & {} & & {} & & {}
\end{tikzcd}$$

Figure 5.3: Tableau for $\dot\exists_x \phi \mathbin{\dot\to} \Box\, y = 1$ of Example 5.5.16

were presented, but the effectiveness of the optimizations still depends on the kind of system (for symbolic representations) or the data-abstraction applied. Moreover, in [52] *tcc* programs were represented in terms of graph structures as a previous work to apply model-checking techniques. These are, to our knowledge, the only adaptation of the model-checking technique to the *ccp* paradigm.

In [45, 93], proof systems to reason about correctness of *tccp* and *ntcc* programs are defined. These works use a temporal logic to define the reactive behavior of programs and to reason about it, but they lack of decision procedures.

5.7 Discussion on the results

In this chapter we have defined an abstract semantics for *tccp* based on a domain of linear temporal formulas with constraints (csLTL) which is sound w.r.t. the behavior of the language.

By using this abstract semantics, we have defined a method to validate csLTL formulas for *tccp* programs. Since the abstract semantics cannot be defined by means of a Galois Connection, we cannot use the abstract diagnosis framework for *tccp* defined in Section 4.1, thus we devised (from scratch) a weak version of the abstract diagnosis framework based only on a concretization function γ. It works by applying $\dot{\mathcal{D}}$ to the abstract specification and then by checking the validity of the resulting implications (whether that computation implies the abstract specification). The computational cost depends essentially on the cost of that check of the implication.

We have also presented an automatic decision procedure for the csLTL logic in order to effectively check the validity of that implication.

Differently from the approach presented in [116], we do not need to restrict the language to the locally-independent fragment since our semantics is able to deal with the full language.

It is worth noticing, that this method does not require to build any model at all, while all the proposals of model checking have in common that a subset of the model of the (target) program has to be built, and sometimes the needed fragment is quite huge. When a property is falsified, model checking provides a counterexample in terms of an erroneous execution trace, leaving to the user the problem of locating the source of the bug. On the contrary, we identify the faulty process declaration.

With our proposal, we can easily specify a possible intervention coming from a surrounding environment simply by a temporal formula. With model checking, this needs to be done by simulating such environment in software with an additional set of declarations.

The approach presented in this chapter encompasses the limitations of the instances presented in the abstract diagnosis formulation of Chapter 4 where the abstract domain used does not allow to specify temporal properties in a straightforward way. In fact, specifications consist of sets of *abstract conditional traces* that are big and onerous to be written. The use of temporal logic certainly overcomes this problem.

In the future, we plan to explore other instances of the method based on logics for which decision procedures or (semi)automatic tools exists. This proposal can also be adapted to other concurrent (non-monotonic) languages (like *tcc* and *ntcc*) once a suitable fully abstract semantics has been developed.

5.A Proofs

5.A.1 Proofs of Section 5.1

Theorem 5.1.2. *Let $D \in \mathbf{D}_{\mathbf{C}}^{\Pi}$ and $\mathcal{S}^\alpha \in \mathbf{I_A}$.*

1. *If there are no abstractly incorrect process declarations in D (i.e., $\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$), then D is partially correct w.r.t. $\mathcal{S}^\alpha$ (i.e., $\mathcal{F}[\![D]\!] \sqsubseteq \gamma(\mathcal{S}^\alpha)$).*

2. *Let D be partially correct w.r.t. $\mathcal{S}^\alpha$. If D has abstract uncovered elements, then D is not complete (i.e., $\gamma(\mathcal{S}^\alpha) \not\sqsubseteq \mathcal{F}[\![D]\!]$).*

Proof of Theorem 5.1.2.

Point 1 By hypothesis, $\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$. Since γ is monotonic $\gamma(\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha}) \sqsubseteq \gamma(\mathcal{S}^\alpha)$. By the soundness of $\mathcal{D}^\alpha$ and by transitivity it follows that $\mathcal{D}[\![D]\!]_{\gamma(\mathcal{S}^\alpha)} \sqsubseteq \gamma(\mathcal{S}^\alpha)$. It can be noticed that $\gamma(\mathcal{S}^\alpha)$ is a pre-fixpoint of $\mathcal{D}[\![D]\!]$, thus, from Knaster-Tarski's theorem it follows directly that $\mathit{lfp}\, \mathcal{D}[\![D]\!] \leq \gamma(\mathcal{S}^\alpha)$. The thesis follows directly from the definition of $\mathcal{F}[\![D]\!]$ ($\mathcal{F}[\![D]\!] = \mathit{lfp}\, \mathcal{D}[\![D]\!]$).

Point 2 By hypothesis, e is such that $e \leq \mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha}$ and $e \wedge \mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} = \bot$. Thus, it follows that $\gamma(\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha}) \sqcap \gamma(\mathcal{S}^\alpha) = \{\epsilon\}$. Since $\mathcal{D}[\![D]\!]_{\gamma(\mathcal{S}^\alpha)} \sqsubseteq \gamma(\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha})$, we have that $\mathcal{D}[\![D]\!]_{\gamma(\mathcal{S}^\alpha)} \sqcap \gamma(\mathcal{S}^\alpha) = \{\epsilon\}$. Suppose that $\gamma(\mathcal{S}^\alpha) \sqsubseteq \mathcal{F}[\![D]\!]$. Since by hypothesis $\mathcal{F}[\![D]\!] \sqsubseteq \gamma(\mathcal{S}^\alpha)$, we have that $\mathcal{F}[\![D]\!] = \gamma(\mathcal{S}^\alpha)$. It follows that $\mathcal{D}[\![D]\!]_{\mathcal{F}[\![D]\!]} \sqcap \mathcal{F}[\![D]\!] = \{\epsilon\}$, but this is a contradiction since $\mathcal{F}$ is a fixpoint. Thus, $\gamma(\mathcal{S}^\alpha) \not\sqsubseteq \mathcal{F}[\![D]\!]$ and the thesis holds.

Theorem 5.1.3. *Let R be a process declaration for $p(\vec{x})$, $\mathcal{S} \in \mathbf{I}$ a concrete specification and $\mathcal{S}^\alpha \in \mathbf{I_A}$ a sound approximation for $\mathcal{S}$ (i.e., $\mathcal{S} \sqsubseteq \gamma(\mathcal{S}^\alpha)$).*

1. *If $\mathcal{D}[\![\{R\}]\!]_{\mathcal{S}} \not\sqsubseteq \gamma(\mathcal{S}^\alpha)$ and it exists e such that $\gamma(e) \sqsubseteq \mathcal{D}[\![\{R\}]\!]_{\mathcal{S}}(p(\vec{x}))$ and $e \wedge \mathcal{S}^\alpha(p(\vec{x})) = \bot$, then R is abstractly incorrect w.r.t. $\mathcal{S}^\alpha$ (on testimony e).*

2. *If there exists an abstract uncovered element e w.r.t. $\mathcal{S}^\alpha$, then there exists $r \in \gamma(e)$ such that $r \notin \mathcal{D}[\![\{R\}]\!]_{\mathcal{S}}(p(\vec{x}))$.*

Proof of Theorem 5.1.3.

Point 1 By hypothesis it exists e such that $\gamma(e) \sqsubseteq \mathcal{D}[\![\{R\}]\!]_{\mathcal{S}}$ and $e \wedge \mathcal{S}^\alpha = \bot$. Since $\mathcal{S} \sqsubseteq \gamma(\mathcal{S}^\alpha)$, we have that $\mathcal{D}[\![\{R\}]\!]_{\mathcal{S}} \sqsubseteq \mathcal{D}[\![\{R\}]\!]_{\gamma(\mathcal{S}^\alpha)}$ and, Since $\mathcal{D}^\alpha$ is a sound approximation of $\mathcal{D}$, $\mathcal{D}[\![\{R\}]\!]_{\gamma(\mathcal{S}^\alpha)} \sqsubseteq \gamma(\mathcal{D}^\alpha[\![\{R\}]\!]_{\mathcal{S}^\alpha})$. The thesis follows directly from the monotonicity of $\mathcal{D}^\alpha$ since $e \leq \mathcal{D}^\alpha[\![\{R\}]\!]_{\mathcal{S}^\alpha}$.

Point 2 By hypothesis $e \wedge \mathcal{D}^\alpha[\![\{R\}]\!]_{\mathcal{S}^\alpha} = \bot$. By the monotonicity of $\mathcal{D}^\alpha$ and since γ is $\bigsqcap$-distributive, it follows that $\gamma(e) \sqcap \gamma(\mathcal{D}^\alpha[\![\{R\}]\!]_{\mathcal{S}^\alpha}) = \{\epsilon\}$. Since $\mathcal{D}^\alpha$ is a sound approximation of $\mathcal{D}$, $\mathcal{D}[\![\{R\}]\!]_{\gamma(\mathcal{S}^\alpha)} \sqsubseteq \gamma(\mathcal{D}^\alpha[\![\{R\}]\!]_{\mathcal{S}^\alpha})$ and, since $\mathcal{S} \sqsubseteq \gamma(\mathcal{S}^\alpha)$, it follows that $\mathcal{D}[\![\{R\}]\!]_{\mathcal{S}} \sqsubseteq \gamma(\mathcal{D}^\alpha[\![\{R\}]\!]_{\mathcal{S}^\alpha})$. Thus, $\gamma(e) \sqcap \mathcal{D}[\![\{R\}]\!]_{\mathcal{S}} = \{\epsilon\}$ and this means that there exists $r \in \gamma(e)$ such that $r \notin \mathcal{D}[\![\{R\}]\!]_{\mathcal{S}}$.

5.A.2 Proofs of Section 5.3

Lemma 5.2.3. *The function $\gamma^{\mathbf{F}}$ is monotonic and injective.*

Proof of Lemma 5.2.3.

$\gamma^{\mathbf{F}}$ **is monotonic.** Let $\phi_1, \phi_2 \in \mathbf{F}$ such that $\phi_1 \dot{\rightarrow} \phi_2$. By Definition 5.2.2, for all $r \in \mathbf{M}$, if $r \vDash \phi_1$ then $r \vDash \phi_2$. Thus, $\bigsqcup\{r \mid r \vDash \phi_1\} \sqsubseteq \bigsqcup\{r \mid r \vDash \phi_2\}$ and, by Equation (5.2.1), $\gamma^{\mathbf{F}}(\phi_1) \sqsubseteq \gamma^{\mathbf{F}}(\phi_2)$.

$\gamma^{\mathbf{F}}$ **is injective.** Let $\phi_1, \phi_2 \in \mathbf{F}$ such that $\gamma^{\mathbf{F}}(\phi_1) = \gamma^{\mathbf{F}}(\phi_2)$. By Equation (5.2.1) and Definition 5.2.2, this means that ϕ_1 and ϕ_2 have the same models, thus, $\phi_1 \dot{\leftrightarrow} \phi_2$.

Lemma 5.3.3. *For each $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$ and each $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$, $\dot{\mathcal{A}}[\![A]\!]$ and $\dot{\mathcal{D}}[\![D]\!]$ are monotonic.*

Proof.

Consider $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$; we show that for each $\dot{\mathcal{I}}_1, \dot{\mathcal{I}}_2 \in \mathbf{I}_{\mathbf{F}}$ and for each $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$, $\dot{\mathcal{I}}_1 \dot{\rightarrow} \dot{\mathcal{I}}_2 \Rightarrow \dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}_1} \dot{\rightarrow} \dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}_2}$. Observe that the only case in which $\dot{\mathcal{A}}$ depends on the interpretation is the case of the process call. By definition of $\dot{\rightarrow}$, $\dot{\mathcal{I}}_1(p(\vec{x})) \dot{\rightarrow} \dot{\mathcal{I}}_2(p(\vec{x}))$, thus:

$$\dot{\mathcal{A}}[\![p(\vec{x})]\!]_{\dot{\mathcal{I}}_1} = \bigcirc \dot{\mathcal{I}}_1(p(\vec{x})) \dot{\rightarrow} \bigcirc \dot{\mathcal{I}}_2(p(\vec{x})) = \dot{\mathcal{A}}[\![p(\vec{x})]\!]_{\dot{\mathcal{I}}_2}$$

The monotonicity of $\dot{\mathcal{D}}[\![D]\!]$ follows directly from the monotonicity of $\dot{\mathcal{A}}[\![A]\!]$.

Lemma 5.A.1 *Given $\phi_1, \phi_2 \in \mathbf{F}$, $\gamma^{\mathbf{F}}(\phi_1 \dot{\wedge} \phi_2) \sqsupseteq \bigsqcup\{r_1 \bar{\|} r_2 \mid r_1 \in \gamma^{\mathbf{F}}(\phi_1),\ r_2 \in \gamma^{\mathbf{F}}(\phi_2),\ r_1 \bar{\|} r_2$ is defined$\}$.*

Proof.

Consider $r_1 \in \gamma^{\mathbf{F}}(\phi_1)$ and $r_2 \in \gamma^{\mathbf{F}}(\phi_2)$ such that $r_1 \bar{\|} r_2$ is defined. We show that $r_1 \bar{\|} r_2 \in \gamma^{\mathbf{F}}(\phi_1 \dot{\wedge} \phi_2)$ (i.e., $r_1 \bar{\|} r_2 \vDash \phi_1 \dot{\wedge} \phi_2$). Since $r_1 \bar{\|} r_2$ is defined, the conditions and stores in r_2 cannot be in contradiction with those in r_1, thus neither with ϕ_1, which means that $(r_1 \bar{\|} r_2) \vDash \phi_1$. Following a similar reasoning, we have that $(r_1 \bar{\|} r_2) \vDash \phi_2$ and finally, from Equation (5.2.2f) we can conclude that $(r_1 \bar{\|} r_2) \vDash \phi_1 \dot{\wedge} \phi_2$.

Lemma 5.A.2 *Given $\phi \in \mathbf{F}$, $\gamma^{\mathbf{F}}(\dot{\exists}_x \phi) \sqsupseteq \bigsqcup\{\bar{\exists}_x r \mid r \in \gamma^{\mathbf{F}}(\phi),\ r$ is x-self-sufficient$\}$.*

Proof.

Let $r \in \gamma^{\mathbf{F}}(\phi)$ be x-self-sufficient. By Equation (5.2.1) $r \vDash \phi$, and by Equation (5.2.2g) it follows directly that $\bar{\exists}_x r \vDash \dot{\exists}_x \phi$. By Equation (5.2.1) we can conclude that $\bar{\exists}_x r \in \gamma^{\mathbf{F}}(\dot{\exists}_x \phi)$.

Theorem 5.3.6. *Let $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$, $D \in \mathbf{D}^{\Pi}_{\mathbf{C}}$ and $\dot{\mathcal{I}} \in \mathbf{I}_{\mathbf{F}}$. Then, $\mathcal{A}[\![A]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})} \sqsubseteq \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}})$ and $\mathcal{D}[\![D]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})} \sqsubseteq \gamma^{\mathbf{F}}(\dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{I}}})$.*

Proof.

Let $A \in \mathbf{A}^{\Pi}_{\mathbf{C}}$ and $\dot{\mathcal{I}} \in \mathbf{I}_{\mathbf{F}}$, we show that $\mathcal{A}[\![A]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})} \sqsubseteq \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}})$ by structural induction on A.

$\underline{A = \mathsf{skip}}\lrcorner$

$$\begin{aligned}
\mathcal{A}[\![\mathsf{skip}]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})} &= \{(\mathit{true},\varnothing) \rightarrowtail \mathit{true} \cdots (\mathit{true},\varnothing) \rightarrowtail \mathit{true} \cdots\} \\
&\qquad [\,\text{by Definition 5.2.2 since } \boxtimes \vDash \mathit{true}\,] \\
&\sqsubseteq \{r \mid r \vDash \dot{\mathit{true}}\} \\
&\qquad [\,\text{by Equation (5.2.1)}\,] \\
&= \gamma^{\mathbf{F}}(\dot{\mathit{true}}) \\
&\qquad [\,\text{by Equation (5.3.1a)}\,] \\
&= \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![\mathsf{skip}]\!]_{\dot{\mathcal{I}}})
\end{aligned}$$

$\underline{A = \mathsf{tell}(c)}\lrcorner$

$$\begin{aligned}
\mathcal{A}[\![\mathsf{tell}(c)]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})} &= \{(\mathit{true},\varnothing) \rightarrowtail c \cdot (c,\varnothing) \rightarrowtail c \cdots (c,\varnothing) \rightarrowtail c \cdots\} \\
&\qquad [\,\text{by Definition 5.2.2}\,] \\
&\sqsubseteq \{r \mid r \vDash \bigcirc c\} \\
&\qquad [\,\text{by Equation (5.2.1)}\,] \\
&= \gamma^{\mathbf{F}}(\bigcirc c) \\
&\qquad [\,\text{by Equation (5.3.1b)}\,] \\
&= \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![\mathsf{tell}(c)]\!]_{\dot{\mathcal{I}}})
\end{aligned}$$

$\underline{A = \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i}\lrcorner$ Let $r \in \mathcal{A}[\![A]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$, we have to prove two cases.

1. Let $r := \underbrace{\mathit{stutt}(\{c_1,\dots,c_n\}) \dots \mathit{stutt}(\{c_1,\dots,c_n\})}_{k \text{ times}} \cdot (c_i,\varnothing) \rightarrowtail c_i \cdot (r_i{\downarrow}_{c_i})$ with $1 \le i \le n$ and $r_i \in \mathcal{A}[\![A_i]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$.
From (5.2.2c) and (5.2.2e), it follows that $\mathit{stutt}(\{c_1,\dots,c_n\}) \vDash \dot{\neg}\, c_i$ for all $1 \le i \le n$. Thus, by (5.2.2f), for all $1 \le j \le k$ $r^j \vDash \dot{\bigwedge}_{i=1}^{n} \dot{\neg}\, c_i$. From (5.2.2c), it follows that $(c_i,\varnothing) \rightarrowtail c_i \vDash c_i$, and, by inductive hypothesis, $r_i \vDash \dot{\mathcal{A}}[\![A_i]\!]_{\dot{\mathcal{I}}}$. Therefore the sub-trace $(c_i,\varnothing) \rightarrowtail c_i \cdot (r_i{\downarrow}_{c_i})$ models the formula $c_i \,\dot{\wedge} \bigcirc \dot{\mathcal{A}}[\![A_i]\!]_{\dot{\mathcal{I}}}$ and as a consequence models also $\dot{\bigvee}_{i=1}^{n} (c_i \,\dot{\wedge} \bigcirc \dot{\mathcal{A}}[\![A_i]\!]_{\dot{\mathcal{I}}})$. Since this sub-trace is preceded in r by the suffix $\mathit{stutt}(\{c_1,\dots,c_n\}) \dots \mathit{stutt}(\{c_1,\dots,c_n\})$, from (5.2.2i), it follows that $r \vDash (\dot{\bigwedge}_{i=1}^{n} \dot{\neg}\, c_i)\, \mathcal{U}\, \dot{\bigvee}_{i=1}^{n} (c_i \,\dot{\wedge} \bigcirc \dot{\mathcal{A}}[\![A_i]\!]_{\dot{\mathcal{I}}})$ and, from (5.3.1c), we can conclude that $r \vDash \dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}}$.
2. Let $r := \mathit{stutt}(\{c_1,\dots,c_n\}) \cdots \mathit{stutt}(\{c_1,\dots,c_n\}) \cdots$. From (5.2.2d) and (5.2.2e) it follows that
$\mathit{stutt}(\{c_1,\dots,c_n\}) \vDash \dot{\neg}\, c_i$ for all $1 \le i \le n$. Thus, by (5.2.2f), $\mathit{stutt}(\{c_1,\dots,c_n\}) \vDash \dot{\bigwedge}_{i=1}^{n} \dot{\neg}\, c_i$. Since r is an infinite replication of $\mathit{stutt}(\{c_1,\dots,c_n\})$, by definition of $\Box$, $r \vDash \Box \dot{\bigwedge}_{i=1}^{n} \dot{\neg}\, c_i$ and, by (5.3.1c) we can conclude that $r \vDash \dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}}$.

$\underline{A = \mathsf{now}\ c\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2}\lrcorner$ Let $r \in \mathcal{A}[\![A]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$. We show that:

$$r \vDash (c \,\dot{\wedge}\, \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}) \,\dot{\vee}\, (\dot{\neg}\, c \,\dot{\wedge}\, \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}).$$

We have to prove seven cases:

1. Let $r := (c, \varnothing) \rightarrowtail c \cdots (c, \varnothing) \rightarrowtail c \cdots$ such that $(\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots \in \mathcal{A}[\![A_1]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$. From (5.2.2c) it follows that $r \vDash c$. By inductive hypothesis, $(\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}})$. Moreover, true is the stronger formula that $(\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots$ can model. Thus, it follows that $\mathit{true} \dot{\rightarrow} \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$. Since $\forall \phi \in \mathsf{csLTL}.\ \phi \dot{\rightarrow} \mathit{true} \dot{\wedge} \phi$, it holds that $r \vDash c \dot{\wedge} \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$.
2. Let $r := (\eta^+ \otimes c, \eta^-) \rightarrowtail d \otimes c \cdot (r' \downarrow_c)$ such that $(\eta^+, \eta^-) \rightarrowtail d \cdot r' \in \mathcal{A}[\![A_1]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$, $d \otimes c \neq \mathit{false}$, $\forall c^- \in \eta^-.\ \eta^+ \otimes c \nvdash c^-$ and r' is c-compatible. By (5.2.2c), it follows that $r \vDash c$. By inductive hypothesis, we know that $(\eta^+, \eta^-) \rightarrowtail d \cdot r' \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}})$, and by (5.2.1), $(\eta^+, \eta^-) \rightarrowtail d \cdot r' \vDash \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$. By hypothesis, $(\eta^+, \eta^-) \rightarrowtail d \cdot r'$ is compatible with c, thus $\dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$ cannot contain $\dot{\neg} c$. Furthermore, it can be noticed that r adds to $(\eta^+, \eta^-) \rightarrowtail d \cdot r'$ only the constraint c in the positive conditions and in the stores, thus, it follows that $r \vDash \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$. By (5.2.2f) we conclude that $r \vDash c \dot{\wedge} \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$.
3. Let $r := (\eta^+ \otimes c, \eta^-) \rightarrowtail \mathit{false} \cdot (\mathit{false}, \varnothing) \rightarrowtail \mathit{false} \cdots (\mathit{false}, \varnothing) \rightarrowtail \mathit{false} \cdots$ such that $(\eta^+, \eta^-) \rightarrowtail d \cdot r' \in \mathcal{A}[\![A_1]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$, $d \otimes c = \mathit{false}$, $\forall c^- \in \eta^-.\ \eta^+ \otimes c \nvdash c^-$ and r' is c-compatible. By (5.2.2c), it follows that $r \vDash c$. By inductive hypothesis, $(\eta^+, \eta^-) \rightarrowtail d \cdot r' \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}})$ and, by (5.2.1), $(\eta^+, \eta^-) \rightarrowtail d \cdot r' \vDash \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$. Reasoning similarly to Point 2 above, it can be noticed that $r \vDash \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$. Thus, by (5.2.2f) we conclude that $r \vDash c \dot{\wedge} \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$.
4. Let $r := (c, \eta^-) \rightarrowtail c \cdot (r' \downarrow_c)$ such that $\mathit{stutt}(\eta^-) \cdot r' \in \mathcal{A}[\![A_1]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$, $\forall c^- \in \eta^-.\ \eta^+ \otimes c \nvdash c^-$ and r' is c-compatible. It follows from (5.2.2c) that $r \vDash c$. By inductive hypothesis, $\mathit{stutt}(\eta^-) \cdot r' \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}})$, and, by (5.2.1), $\mathit{stutt}(\eta^-) \cdot r' \vDash \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$. Reasoning as in Point 2 of this proof it follows that $r \vDash \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$. Thus, $r \vDash c \dot{\wedge} \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$.
5. Let $r := (\mathit{true}, \{c\}) \rightarrowtail \mathit{true} \cdot (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots$ such that $(\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots \in \mathcal{A}[\![A_2]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$. By (5.2.2c) and (5.2.2e), $r \vDash \dot{\neg} c$. By inductive hypothesis, $(\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdots \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}})$ and, reasoning as in Point 1 of this proof, it follows that $\mathit{true} \dot{\rightarrow} \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$. Since $\forall \phi \in \mathsf{csLTL}.\ \phi \dot{\rightarrow} \mathit{true} \dot{\wedge} \phi$, it holds that $r \vDash \dot{\neg} c \dot{\wedge} \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$.
6. Let $r := (\eta^+, \eta^- \cup \{c\}) \rightarrowtail d \cdot r'$ such that $(\eta^+, \eta^-) \rightarrowtail d \cdot r' \in \mathcal{A}[\![A_2]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$ and $\eta^+ \nvdash c$. By (5.2.2c), $r \vDash \dot{\neg} c$. By inductive hypothesis, $(\eta^+, \eta^-) \rightarrowtail d \cdot r' \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}})$ and, by (5.2.1), $(\eta^+, \eta^-) \rightarrowtail d \cdot r' \vDash \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$. It can be noticed that $\dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$ cannot imply the formula c, otherwise $(\eta^+, \eta^-) \rightarrowtail d \cdot r'$ would not be a model for $\dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$ since by hypothesis $\eta^+ \nvdash c$. Since r differs from $(\eta^+, \eta^-) \rightarrowtail d \cdot r'$ only in the presence of c in the first negative condition, it follows that $r \vDash \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$. Thus, by (5.2.2f) we conclude that $r \vDash \dot{\neg} c \dot{\wedge} \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$.
7. Let $r := (\mathit{true}, \eta^- \cup \{c\}) \rightarrowtail \mathit{true} \cdot r'$ such that $\mathit{stutt}(\eta^-) \cdot r' \in \mathcal{A}[\![A_2]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$. By (5.2.2d), $r \vDash \dot{\neg} c$ and, by inductive hypothesis, $\mathit{stutt}(\eta^-) \cdot r' \vDash \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$. Reasoning as in the previous Point 6 it can be noticed that $r \vDash \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$ and, therefore, $r \vDash \dot{\neg} c \dot{\wedge} \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$.

We have proven that for all $r \in \mathcal{A}[\![A]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$ either $r \vDash c \dot{\wedge} \dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}}$ or $r \vDash \dot{\neg} c \dot{\wedge} \dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}}$. Therefore, from (5.3.1d) we conclude that $r \vDash \dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}}$.

$\underline{\boldsymbol{A = A_1 \parallel A_2}}\lrcorner$ Let $r_1 \bar{\parallel} r_2 \in \mathcal{A}[\![A_1 \parallel A_2]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$ such that $r_1 \in \mathcal{A}[\![A_1]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$ and $r_2 \in \mathcal{A}[\![A_2]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$. By inductive hypothesis, $r_1 \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}})$ and $r_2 \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A_2]\!]_{\dot{\mathcal{I}}})$. It follows from Lemma 5.A.1 that $r_1 \bar{\parallel} r_2 \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A_1 \parallel A_2]\!]_{\dot{\mathcal{I}}})$.

$\underline{\boldsymbol{A = \exists x\, A_1}}\lrcorner$ Let $\bar{\exists}_x\, r_1 \in \mathcal{A}[\![\exists x\, A_1]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$ such that $r_1 \in \mathcal{A}[\![A_1]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$ and r_1 is x-self-sufficient. By inductive hypothesis, $r_1 \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A_1]\!]_{\dot{\mathcal{I}}})$ and, by Lemma 5.A.2, it follows that $\bar{\exists}_x\, r_1 \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![\exists x\, A_1]\!]_{\dot{\mathcal{I}}})$.

$\underline{\boldsymbol{A = p(\vec{x})}}\lrcorner$ Let $r := (\mathit{true}, \varnothing) \rightarrowtail \mathit{true} \cdot r' \in \mathcal{A}[\![p(\vec{x})]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}$ such that $r' \in \gamma^{\mathbf{F}}(\dot{\mathcal{I}}(p(\vec{x})))$. By (5.2.1), it follows that $r' \vDash \dot{\mathcal{I}}(p(\vec{x}))$. From (5.2.2h) we can conclude that $r \vDash \bigcirc \dot{\mathcal{I}}(p(\vec{x}))$ and, thus, $r \in \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![p(\vec{x})]\!]_{\dot{\mathcal{I}}})$.

Let $D \in \mathbf{D}_{\mathbf{C}}^{\Pi}$ and $\dot{\mathcal{I}} \in \mathbf{I}_{\mathbf{F}}$. We prove that $\mathcal{D}[\![D]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})} \sqsubseteq \gamma^{\mathbf{F}}(\dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{I}}})$ by showing that for all $p(x) \mathrel{:-} A \in D$, $\mathcal{D}[\![D]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}(p(\vec{x})) \sqsubseteq \gamma^{\mathbf{F}}(\dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{I}}}(p(\vec{x})))$.

$$\begin{aligned}
\mathcal{D}[\![D]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})}(p(\vec{x})) &= \bigsqcup_{p(x) :- A \in D} \mathcal{A}[\![A]\!]_{\gamma^{\mathbf{F}}(\dot{\mathcal{I}})} \\
&\quad [\text{by the soundness of } \dot{\mathcal{A}}\,] \\
&\sqsubseteq \bigsqcup_{p(x) :- A \in D} \gamma^{\mathbf{F}}(\dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}}) \\
&\quad [\text{by the monotonicity of } \gamma^{\mathbf{F}} \text{ (Lemma 5.2.3)}\,] \\
&\sqsubseteq \gamma^{\mathbf{F}}(\dot{\bigvee}_{p(x) :- A \in D} \dot{\mathcal{A}}[\![A]\!]_{\dot{\mathcal{I}}}) \\
&= \gamma^{\mathbf{F}}(\dot{\mathcal{D}}[\![D]\!]_{\dot{\mathcal{I}}})(p(\vec{x}))
\end{aligned}$$

5.A.3 Proofs of Section 5.5

In this section we present the proofs of the results presented in Section 5.5 together with some auxiliary definitions and results which are used in those proofs.

We first show that α- and β-formulas rules and the next operator preserve the satisfiability of a set of formulas.

Lemma 5.5.1. *Given a set of formulas Φ, an α-formula α and a β-formula β:*

1. $\Phi \cup \{\alpha\}$ *is satisfiable* $\Longleftrightarrow$ $\Phi \cup A(\alpha)$ *is satisfiable;*
2. $\Phi \cup \{\beta\}$ *is satisfiable* $\Longleftrightarrow$ $\Phi \cup B_1(\beta)$ *or* $\Phi \cup B_2(\beta)$ *is satisfiable;*
3. *if Φ is a set of elementary formulas, Φ is satisfiable* $\Longleftrightarrow$ $\mathsf{next}(\Phi)$ *is satisfiable;*

Proof.

We prove the three points separately.

1. Let us consider the rules for α-formulas in Figure 5.1. Let Φ be a set of formulas, α an α-formula and $\phi, \phi_1, \phi_2 \in \mathsf{csLTL}$.

   $\underline{\mathbf{R1}}\lrcorner$ Let $\alpha = \dot{\neg}\,\dot{\neg}\,\phi$, this case follows directly from the equivalence $\dot{\neg}\,\dot{\neg}\,\phi = \phi$.

R2 Let $\alpha = \phi_1 \dot{\wedge} \phi_2$, this case follows directly from Definition 5.2.2, in particular Equation (5.2.2f).

R3 Let $\alpha = \dot{\neg} \bigcirc \phi$, this case follows directly from the equivalence $\dot{\neg} \bigcirc \phi = \bigcirc \dot{\neg} \phi$.

2. Let us consider the rules for β-formulas in Figure 5.1. Let Φ be a set of formulas, β a β-formula and $\phi_1, \phi_2 \in \mathsf{csLTL}$.

R4 Let $\beta = \dot{\neg}(\phi_1 \dot{\wedge} \phi_2)$. We show the two directions independently.

$\Rightarrow$ Assume that it exists $r \in \mathbf{M}$ such that $r \vDash \Phi \cup \{\dot{\neg}(\phi_1 \dot{\wedge} \phi_2)\}$. By applying De Morgan laws $r \vDash \Phi \cup \{\dot{\neg} \phi_1 \dot{\vee} \dot{\neg} \phi_2\}$. By Definition 5.2.2 it follows directly that $r \vDash \Phi \cup \{\dot{\neg} \phi_1\}$ or $r \vDash \Phi \cup \{\dot{\neg} \phi_2\}$.

$\Leftarrow$ Without lost of generality assume that it exists $r \in \mathbf{M}$ such that $r \vDash \Phi \cup \{\dot{\neg} \phi_1\}$. It follows that $r \vDash \Phi \cup \{\dot{\neg} \phi_1 \dot{\vee} \dot{\neg} \phi_2\}$ and by De Morgan laws $r \vDash \Phi \cup \{\dot{\neg}(\phi_1 \dot{\wedge} \phi_2)\}$.

R5 Let $\beta = \dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)$.

$\Rightarrow$ Assume that it exists $r \in \mathbf{M}$ such that $r \vDash \Phi \cup \{\dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)\}$. We build a model for at least one of the following sets: $\Phi \cup \{\phi_1, \dot{\neg} \phi_2, \dot{\neg} \bigcirc(\phi_1 \,\mathcal{U}\, \phi_2)\}$ and $\Phi \cup \{\dot{\neg} \phi_1, \dot{\neg} \phi_2\}$. We distinguish two cases.
In case $r \vDash \phi_1$, we have $r \vDash \Phi \cup \{\phi_1, \dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)\}$, thus, by the fixpoint characterization of $\mathcal{U}$, $r \vDash \Phi \cup \{\phi_1, \dot{\neg}(\phi_2 \dot{\vee} \bigcirc(\phi_1 \,\mathcal{U}\, \phi_2))\}$. It can be notice that $\dot{\neg}(\phi_2 \dot{\vee} \bigcirc(\phi_1 \,\mathcal{U}\, \phi_2)) = \dot{\neg} \phi_2 \dot{\wedge} \dot{\neg} \bigcirc(\phi_1 \,\mathcal{U}\, \phi_2)$ and by Definition 5.2.2 it follows that $r \vDash \Phi \cup \{\phi_1, \dot{\neg} \phi_2, \dot{\neg} \bigcirc(\phi_1 \,\mathcal{U}\, \phi_2)\}$.
Otherwise, in case $r \nvDash \phi_1$, $r \vDash \Phi \cup \{\dot{\neg} \phi_1, \dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)\}$. This means that $r \vDash \dot{\neg} \phi_1$, $r \vDash \dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)$ and $r \vDash \Phi$. By definition of $\mathcal{U}$ it follows that $r \nvDash \phi_2$, otherwise $r \vDash \phi_1 \,\mathcal{U}\, \phi_2$ and this contradicts the hypothesis. Therefore, $r \vDash \dot{\neg} \phi_1$ and $r \vDash \dot{\neg} \phi_2$ and we can conclude that $r \vDash \Phi \cup \{\dot{\neg} \phi_1, \dot{\neg} \phi_2\}$.

$\Leftarrow$ Assume that it exists $r \in \mathbf{M}$ such that $r \vDash \Phi \cup \{\phi_1, \dot{\neg} \phi_2, \dot{\neg} \bigcirc(\phi_1 \,\mathcal{U}\, \phi_2)\}$. By definition of $\mathcal{U}$ if follows that $r \nvDash \phi_1 \,\mathcal{U}\, \phi_2$, since ϕ_2 and $\bigcirc(\phi_1 \,\mathcal{U}\, \phi_2)$ are not modeled by r. Thus, we can conclude that $r \vDash \Phi \cup \{\dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)\}$.
Now assume that it exists $r \in \mathbf{M}$ such that $r \vDash \Phi \cup \{\dot{\neg} \phi_1, \dot{\neg} \phi_2\}$. Since neither ϕ_1 nor ϕ_2 are not modeled by r, it follows that $r \nvDash \phi_1 \,\mathcal{U}\, \phi_2$, thus, $r \vDash \Phi \cup \{\dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)\}$.

R6 Let $\beta = \phi_1 \,\mathcal{U}\, \phi_2$ be an eventuality in the context Φ.

$\Rightarrow$ Assume that it exists $r \in \mathbf{M}$ such that $r \vDash \Phi \cup \{\phi_1 \,\mathcal{U}\, \phi_2\}$, we build a model for at least one of the following sets: $\Phi \cup \{\phi_2\}$ and $\Phi \cup \{\phi_1, \dot{\neg} \phi_2, \bigcirc((\Phi^* \dot{\wedge} \phi_1) \,\mathcal{U}\, \phi_2)\}$. Let $j \geq 0$ be the least j such that $r^j \vDash \phi_2$. If $j = 0$ then $r \vDash \phi_2$ and $r \vDash \Phi \cup \{\phi_2\}$. Otherwise, if $j > 0$, then $r \nvDash \phi_2$ and, by definition of $\mathcal{U}$, $r \vDash \phi_1$. Let i be the greatest index such that $0 \leq i < l$ and $r^i \vDash \Phi \cup \{\phi_1 \,\mathcal{U}\, \phi_2\}$. It follows that Φ or $\phi_1 \,\mathcal{U}\, \phi_2$ should not hold in the next time instant. Since ϕ_2 has not be reached yet we have that $r^{i+1} \vDash \phi_1 \,\mathcal{U}\, \phi_2$, thus, at least one $\phi \in \Phi$ should not be modeled by r^{i+1}. It follows that $r^i \vDash \bigcirc((\Phi^* \dot{\wedge} \phi_1) \,\mathcal{U}\, \phi_2)$.

$\Leftarrow$ We have to distinguish two cases. Assume that it exists $r \in \mathbf{M}$ such that $r \vDash \Phi \cup \{\phi_2\}$, thus, $r \vDash \Phi \cup \{\phi_1 \,\mathcal{U}\, \phi_2\}$.
Otherwise assume that it exists $r \in \mathbf{M}$ such that $r \vDash \Phi \cup \{\bigcirc((\Phi^* \dot{\wedge} \phi_1) \,\mathcal{U}\, \phi_2), \phi_1, \dot{\neg} \phi_2\}$. Since $r \vDash \bigcirc((\Phi^* \dot{\wedge} \phi_1) \,\mathcal{U}\, \phi_2)$ we have that $r \vDash \bigcirc(\phi_1 \,\mathcal{U}\, \phi_2)$. Thus, by definition of $\mathcal{U}$ we conclude that $r \vDash \Phi \cup \{\phi_1 \,\mathcal{U}\, \phi_2\}$.

3. Consider the set $\Phi = \{c_1, \dots, c_n, \bigcirc\phi_1, \dots, \bigcirc\phi_m, \dot{\neg}\bigcirc\psi_1, \dots, \dot{\neg}\bigcirc\psi_k\}$, with $c_1, \dots, c_n \in \mathcal{C}$ and $\phi_1, \dots, \phi_m, \psi_1, \dots, \psi_k \in \mathsf{csLTL}$. We show the two directions independently.

   $\Rightarrow$ Assume that it exists $r \in \mathbf{M}$ such that $r \vDash \Phi$. Let us recall that r^1 is the suffix of r obtained by deleting the first element of r. By Definition 5.2.2 it follows that $r \vDash c_i$ for $i = 1 \dots n$, $r \vDash \bigcirc\phi_j$ for $j = 1 \dots m$ and $r \nvDash \bigcirc\psi_l$ for $l = 1 \dots k$. From monotonicity of r it follows that $r^1 \vDash c_i$ for $i = 1 \dots n$. Moreover, by (5.2.2h), $r^1 \vDash \phi_j$ for $j = 1 \dots m$ and $r^1 \nvDash \psi_l$ for $l = 1 \dots k$. Thus it follows directly that $r^1 \vDash \mathsf{next}(\Phi)$.

   $\Leftarrow$ Now assume that it exists $r \in \mathbf{M}$ such that $r \vDash \mathsf{next}(\Phi)$. Consider $C \coloneqq c_1 \otimes \cdots \otimes c_n$. It is easy to notice that $r' \coloneqq (C, \varnothing) \rightarrowtail C \cdot r$ is a monotone and consistent conditional trace, otherwise $r(1) \nvDash c$ and $r \nvDash \mathsf{next}(\Phi)$. We show that $(C, \varnothing) \rightarrowtail C \cdot r$ is a model for Φ. By definition of C, is easy to notice that $(C, \varnothing) \rightarrowtail C \vDash c_i$ for $i = 1 \dots n$. Furthermore, by (5.2.2h), $(C, \varnothing) \rightarrowtail C \cdot r \vDash \bigcirc\phi_j$ for $j = 1 \dots m$ and $r \nvDash \psi_l$ for $l = 1 \dots k$, thus $(C, \varnothing) \rightarrowtail C \cdot r \nvDash \dot{\neg}\bigcirc\psi_l$. Therefore, $(C, \varnothing) \rightarrowtail C \cdot r \vDash \Phi$.

The correctness of the rule for the existential quantification derives from the following lemma, which shows that $\dot{\exists}_x\,\phi$ and ϕ are equi-satisfiable.

Lemma 5.5.2. *Let $\phi \in$ csLTL, $\dot{\exists}_x\,\phi$ is satisfiable $\iff$ ϕ satisfiable.*

Proof.

We show the two directions independently.

$\Rightarrow$ This direction follows directly from (5.2.2g).

$$\begin{aligned}
\dot{\exists}_x\,\phi \text{ satisfiable} &\Rightarrow \text{it exists } r \in \mathbf{M}.\ r \vDash \dot{\exists}_x\,\phi \\
&\Rightarrow \text{it exists } r' \in \mathbf{M}.\ \bar{\exists}_x\, r' = \bar{\exists}_x\, r \text{ and } r' \vDash \phi \\
&\Rightarrow \phi \text{ satisfiable}
\end{aligned}$$

$\Leftarrow$ Let r be a model for ϕ, if we remove from r the information regarding x, we obtain a model $r' \coloneqq \bar{\exists}_x\, r$ for $\dot{\exists}_x\,\phi$. Indeed, $\bar{\exists}_x\, r = \exists_x\, r$ ($\exists$ is idempotent) and $r \vDash \phi$, thus, by (5.2.2g) $r' \vDash \dot{\exists}_x\,\phi$.

Corollary 5.5.3. *Let $\Phi \subseteq$ csLTL such that $x \in Var$ does not appear in Φ and let $\phi \in$ csLTL. Then, $\Phi \cup \{\dot{\exists}_x\,\phi\}$ is satisfiable $\iff$ $\Phi \cup \{\phi\}$ is satisfiable.*

Proof.

Follows directly from Lemma 5.5.2. x does not appear in Φ, thus the local variable x of ϕ is independent from any other variable in Φ.

In Subsection 5.5.2 we gave the informal definitions of path and stages in a tableau. In the following we formally define these notions.

Definition 5.A.3 *Let $b = n_0, n_1, \dots n_k$ be an open branch such that $L(n_k) = L(n_j)$ for $0 \le j < k$, then b is cyclic and we define* $\mathsf{path}(b) = n_0, n_1, \dots, n_j, (n_{j+1}, \dots, n_k)^\omega$.

Definition 5.A.4 *Given a branch b, every maximal subsequence $n_i, n_{i+1}, \dots n_j$ of* $\mathsf{path}(b)$ *is called a* stage *if, for all $i \leq l \leq j$, $L(n_l)$ is not formed only by elementary formulas or $L(n_l) \neq \mathsf{next}(L(n_{l-1}))$. We denote by* $\mathsf{stages}(b)$ *the sequence of the stages in b.*

We distinguish a particular class of stages called saturated.

Definition 5.A.5 *A stage s is saturated if and only if for every $\phi \in L(s)$:*

- *if ϕ is an α-formula then $A(\alpha) \subseteq L(s)$;*
- *if ϕ is an beta-formula then $B_1(\beta) \subseteq L(s)$ or $B_2(\beta) \subseteq L(s)$;*
- *if $\phi = \dot{\exists}_x \phi'$ with $x \in Var$ and $\phi' \in \mathsf{csLTL}$ then $\phi' \in L(s)$.*

Definition 5.A.6 *Let $\mathcal{T}_\Phi$ be a tableau and $S = s_0, s_1, \dots, s_n$ be a sequence of stages in $\mathcal{T}_\Phi$. Any eventuality $\phi_1 \,\mathcal{U}\, \phi_2 \in L(s_i)$ with $0 \leq i \leq n$ is said to be* fulfilled *in S if there exists $j \geq i$ such that $\phi_2 \in L(s_j)$.*

Intuitively, the formula is fulfilled in the path if we can reach (following the path) a node where ϕ_2 is true.

Definition 5.A.7 *A sequence of stages S is said to be* fulfilling *if and only if every eventuality occurring in S is fulfilled in S. A branch b is said to be* fulfilling *if and only if* $\mathsf{path}(\mathsf{stages}(b))$ *is fulfilling.*

Now we give the definition of expanded branch. Open expanded branches correspond to models of the initial set of formulas.

Definition 5.A.8 *An open branch b is* expanded *if and only if b is fulfilling and each stage in* $\mathsf{stages}(b)$ *is saturated.*

When constructing a tableau only non-expanded open branches are selected to be enlarged with the rules in Figure 5.1. When all branches are closed or expanded the tableau cannot be further expanded.

Proposition 5.A.9 *Let $\mathcal{T}_\Phi$ be the systematic tableau for Φ, each stage s occurring in $\mathcal{T}_\Phi$ is saturated.*

Proof.

By looking into Definition 5.5.9 it can be noticed that the algorithm applies any possible α-, β-rule and $\dot{\exists}$ elimination before applying the next operator to jump to the following stage.

It can be proven that starting from a finite set of formulas Φ, the set of formulas which can occur in the construction of the systematic tableau $\mathcal{T}_\Phi$ is finite. This result is the adaptation to the csLTL case of the corresponding result for PLTL shown in [60].

We denote as $clo(\Phi)$ the closure of a set of formulas Φ which contains all the formulas that can occur in any systematic tableau for Φ.

Let us first introduces some auxiliary sets of formulas which are used in the definition of $clo(\Phi)$.

We denote as $\mathit{subf}(\Phi)$ the set of sub-formulas in Φ and their negations. $\mathit{preclo}(\Phi)$ extends $\mathit{subf}(\Phi)$ with the formulas that can be generated from $\mathit{subf}(\Phi)$ by means of the rules in Figure 5.1 (α, β rules and $\dot\exists$ elimination) except Rule R6.

$$\begin{aligned}\mathit{preclo}(\Phi) := \mathit{subf}(\Phi) \cup \{&\bigcirc(\phi_1 \,\mathcal{U}\, \phi_2), \dot\neg \bigcirc(\phi_1 \,\mathcal{U}\, \phi_2), \bigcirc \dot\neg(\phi_1 \,\mathcal{U}\, \phi_2) \mid \phi_1 \,\mathcal{U}\, \phi_2 \in \mathit{subf}(\Phi)\} \\ &\{\bigcirc(\dot\neg\, \phi) \mid \dot\neg(\bigcirc \phi) \in \mathit{subf}(\Phi)\} \cup \{\phi \mid \dot\exists_x\, \phi \in \mathit{subf}(\Phi)\}\end{aligned}$$

$\mathit{clo}(\Phi)$ captures the formulas generated by Rule R6 by using $\mathit{negctx}(\Phi)$ which represents the conjunctions of negated contexts introduced by Rule R6.

$$\begin{aligned}\mathit{clo}(\Phi) :=& \big\{\dot{\bigwedge}\, \Delta \mid \Delta \subseteq \{\phi_1 \mid \phi_1 \,\mathcal{U}\, \phi_2 \in \mathit{subf}(\Phi)\} \cup \mathit{negctx}(\Phi)\big\} \\ &\text{where } \ \mathit{negctx}(\Phi) := \{\Gamma^* \mid \Gamma \subseteq \mathit{preclo}(\Phi)\}\end{aligned}$$

Definition 5.A.10 *Let Φ be a set of formulas, the closure of Φ is defined as*

$$\begin{aligned}\mathit{clo}(\Phi) :=\,& \mathit{preclo}(\Phi) \cup \mathit{clo}(\Phi) \\ &\cup \{(\phi_1 \,\dot\wedge\, \phi_2) \,\mathcal{U}\, \psi, \bigcirc((\phi_1 \,\dot\wedge\, \phi_2) \,\mathcal{U}\, \psi) \mid \phi \,\mathcal{U}\, \psi \in \mathit{subf}(\Phi) \ \text{ and } \phi_1, \phi_2 \in \mathit{clo}(\Phi)\}\end{aligned}$$

Proposition 5.A.11 *Let $\Phi \subseteq$ csLTL be a finite set, then $\mathit{clo}(\Phi)$ is also finite.*

Proof.

It follows directly from Definition 5.A.10.

The fact that $\mathit{clo}(\Phi)$ is finite is not enough to guarantee that the algorithm terminates in a finite number of steps. It is necessary to assume that the algorithm uses a *fair strategy* to distinguish eventualities. this means that no eventuality formula in an open branch can remain non-distinguished indefinitely. A fair strategy guarantees the termination of the construction.

Let us recall some significant results shown in [60] about the handling of eventualities in the construction of the systematic tableau $\mathcal{T}_\Phi$ for a set of formulas Φ.

Proposition 5.A.12 *Let s be a stage in a branch b of $\mathcal{T}_\Phi$, if $\{\phi, \dot\neg\, \phi\} \subseteq L(s)$ then every branch prefixed by b is closed.*

Proof.

It can be noticed that the application of the rules in Figure 5.1 to two complementary formulas belonging to the same stage (but not necessarily to the same node) will generate two complementary formulas that belong to the same node.

The following proposition states that non-satisfied undistinguished eventualities are kept in branches at least until they are fulfilled or they become distinguished.

Proposition 5.A.13 *Let b be a branch of $\mathcal{T}_\Phi$ and $s_0, s_1, \ldots, s_k$ be a prefix of* $\mathsf{path}(\mathsf{stages}(b))$. *If $\phi_1 \,\mathcal{U}\, \phi_2 \in L(n_i)$ for some $0 \le i \le k$, $\phi_1 \,\mathcal{U}\, \phi_2$ is not distinguished in $s_i, \ldots, s_k$ and $\phi_2 \notin L(s_i) \cup \cdots \cup L(s_k)$, then $\{\phi_1, \dot\neg\, \phi_2, \bigcirc(\phi_1 \,\mathcal{U}\, \phi_2)\} \subseteq L(s_j)$ for all $i \le j \le k$.*

Proof.

By the construction of $\mathcal{T}_\Phi$ since undistinguished eventualities are handled by Rule R6.

The following proposition states that if a distinguished eventuality $\phi_1 \,\mathcal{U}\, \phi_2$ is not fulfilled in and expanded branch b, then b is closed, since the expansion of $\phi_1 \,\mathcal{U}\, \phi_2$ by using Rule R6, is in contradiction with the context.

Proposition 5.A.14 *Let b be a branch of $\mathcal{T}_\Phi$ and $s_0, s_1, \ldots, s_k$ be a prefix of* $\mathsf{path}(\mathsf{stages}(b))$. *Consider the eventuality $\phi_1 \,\mathcal{U}\, \phi_2$, and let i be the least index such that the eventuality $\phi_1 \,\mathcal{U}\, \phi_2$ is distinguished in the stage s_i. If $\phi_2 \notin L(s_i) \cup \cdots \cup L(s_k)$ then, for all $0 \leq l \leq k - i$, $\{\delta_l, \dot{\neg}\, \phi_2, \bigcirc(\delta_{l+1} \,\mathcal{U}\, \phi_2)\} \subseteq L(s_{i+l})$ where $\delta_0 = \phi_1$ and $\delta_{l+1} = \delta_l \mathrel{\dot{\wedge}} \chi$ for some $\chi \in negctx(\Phi)$.*

Moreover, if $\delta_l = \dot{\bigwedge}\Gamma$ for some Γ such that $\chi \in \Gamma$, then every maximal branch prefixed by $s_0, \ldots, s_{i+l}$ is closed.

Proof.

By construction of $\mathcal{T}_\Phi$, distinguished eventualities are handled by Rule R6. This rule gives rise to two branches: one containing $\{\gamma_l, \dot{\neg}\, \phi_2, \bigcirc(\gamma_{l+1} \,\mathcal{U}\, \phi_2)\}$ and the other containing ϕ_2. If $\bigcirc(\gamma_{l+1} \,\mathcal{U}\, \phi_2)$ is the distinguish eventuality in a successive node n in stage s_{i+l} then, in the next stage, s_{i+l} the distinguished eventuality is $\gamma_{l+1} \,\mathcal{U}\, \phi_2$ in a node n'. By Rule R6, $\gamma_0 = \phi_1$ and for all $j > 0$ $\gamma_j = \gamma_{j-1} \mathrel{\dot{\wedge}} \Delta^*_{j-1}$ where $\Delta^*_{j-1} \in negctx(\Phi)$ and Γ_{j-1} is the context, $L(n) \setminus \{\bigcirc(\gamma_{l+1} \,\mathcal{U}\, \phi_2)\}$. Therefore, by induction on l, $\gamma_l \in clo(\Phi)$ for all $0 \leq l \leq k-1$.

Moreover we have that χ is the negation of the context of a node in s_{i+l}, if $\delta_l = \dot{\bigwedge}\Gamma$ for some Γ such that $\chi \in \Gamma$, then every branch prefixed by $s_0, \ldots, s_{i+l}$ contains at the same stage two complementary formulas $\{\psi, \dot{\neg}\, \psi\}$. From Proposition 5.A.12 we can conclude every maximal branch prefixed by $s_0, \ldots, s_{i+l}$ is closed.

Corollary 5.A.15 *Every distinguish eventuality in a cyclic branch of $\mathcal{T}_\Phi$ is fulfilled.*

Proof.

By Proposition 5.A.14 if a distinguish eventuality in a branch b is unfulfilled, then b is closed and it is not cyclic.

Proposition 5.A.16 *Let b be a branch of $\mathcal{T}_\Phi$. b is open if and only if one of the following points holds:*

1. *the last node of b contains only constraint formulas;*
2. *b is cyclic and for every eventuality $\phi \in L(n)$ for a node occurring in b, ϕ is fulfilled in b.*

Proof.

It follows directly from Point 3 and Point 6a in the algorithm of Definition 5.5.9 and from Proposition 5.A.12 and Corollary 5.A.15.

Let us recall Lemma 5.5.10.

Lemma 5.5.10. *The algorithm of Definition 5.5.9 when using a fair strategy for the selection of eventualities, given as input a finite set $\Phi \subseteq$ csLTL, terminates and builds an expanded tableau for $\mathcal{T}_\Phi$.*

Proof.

Suppose that the algorithm does not terminate. This means that $\mathcal{T}_\Phi$ contains an infinite branch $b = n_1, n_2, \ldots, n_i \ldots$. By Propositions 5.A.11, 5.A.14 and 5.A.16 this can happen only if b contains an eventuality that is never distinguished, which contradicts the fairness assumption.

The following proposition shows the behavior of negated eventualities. It is needed to prove completeness.

Proposition 5.A.17 *Let b be a branch in the systematic tableau $\mathcal{T}_\Phi$ for $\Phi \subseteq$ csLTL, and let s_j be a stage of the path p in the branch ($p =$ path(b)) such that $\dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2) \in L(s_j)$. Then, every finite subsequence of p of the form $\pi = s_j, s_{j+1}, \ldots, s_k$ satisfies one of the following properties:*

1. *$\{\phi_1, \dot{\neg}\phi_2, \bigcirc\dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)\} \subseteq L(s_i)$ for $j \le i \le k$.*
2. *There exists $j \le i \le k$ such that $\{\dot{\neg}\phi_1, \dot{\neg}\phi_2\} \subseteq L(s_i)$ and $\{\phi_1, \dot{\neg}\phi_2, \bigcirc\dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)\} \subseteq L(s_l)$ for $j \le l \le i-1$.*

Proof.

We proceed by induction of $k - j$. In case $k = j$ the property follows directly from Rule R5 and since each stage of a systematic tableau is saturated (Proposition 5.A.9). In case $k > j$, by inductive hypothesis we have that $\pi' = s_j, \ldots, s_{k-1}$ satisfies one of the two properties of Proposition 5.A.17. If π' satisfies Point 1 then by the saturation of the stage (Proposition 5.A.9) it follows that $\{\phi_1, \dot{\neg}\phi_2, \dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)\} \subseteq L(s_k)$ or $\{\dot{\neg}\phi_1, \dot{\neg}\phi_2\} \subseteq L(s_k)$, thus π verifies Point 1 or Point 2 respectively. Otherwise, if π' satisfies Point 2 so does π.

This proposition ensures that, if a node is labeled with a negated eventuality, then every node in a finite suffix of the path from that node, by construction, does not contain the second part of the eventuality (ϕ_2).

Given the function stores defined in Subsection 5.5.4, we show that, from a systematic tableau $\mathcal{T}_\Phi$ built for Φ, we can compute a model for Φ from every open branch b in $\mathcal{T}_\Phi$.

Lemma 5.5.12. *Let b be an* open expanded *branch in the systematic tableau $\mathcal{T}_\Phi$ for $\Phi \subseteq$ csLTL. Given the sequence of stages S in* path(b), *then* stores$(S) \models \Phi$.

Proof.

Let $r :=$ stores(S). To show that $r \models \Phi$, it is sufficient to show that for all $\phi \in \Phi$, $r \models \phi$. Note that, by Definition 5.5.9 and by the definition of stores, r contains, at each time instant, all the constraints in the labeling of the nodes in the corresponding stage. We proceed by induction on the structure of ϕ.

- Let $\phi = c$ with $c \in \mathbf{C}$; Since the first state in r contains c (which we know belongs to the labels in the first stage), then by the definition of $\models$ (Definition 5.2.2), $r \models c$.
- Let ϕ be of one of the following forms $\dot{\neg}\dot{\neg}\phi_1$, $\phi_1 \dot{\wedge} \phi_2$, $\dot{\neg}\phi_1 \dot{\wedge} \phi_2$, $\bigcirc\phi_1$, $\dot{\neg}\bigcirc\phi_1$ or $\dot{\exists}_x \phi_1$; Since every stage is saturated and by induction hypothesis on $\{\phi_1\}$, $\{\phi_1, \phi_2\}$, $\{\dot{\neg}\phi_1, \dot{\neg}\phi_2\}$, $\{\phi_1\}$, $\{\dot{\neg}\phi_1\}$ and $\{\phi_1\}$, respectively, $r \models \phi$.

- Let $\phi = \phi_1 \,\mathcal{U}\, \phi_2$, since b is an open extended branch, ϕ is fulfilled in b and, as a consequence, in $\mathsf{path}(S)$. Therefore, it exists a finite subsequence $s_0, s_1, \ldots, s_n$ of $\mathsf{path}(S)$ such that $\phi_2 \in L(s_n)$ and for all $0 \leq i < n$, $\phi_1 \in L(s_i)$. By inductive hypothesis, $r^n \vDash \phi_2$ and for all $0 \leq i < n$, $r^i \vDash \phi_1$. By (5.2.2i) in Definition 5.2.2, it follows that $r \vDash \phi_1 \,\mathcal{U}\, \phi_2$.

- Let $\phi = \dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)$ By Proposition 5.A.17 it does not exist a finite subsequence $s_0, s_1, \ldots, s_n$ of $\mathsf{path}(\mathsf{stages}(b))$ such that $\phi_2 \in L(s_n)$ and for all $0 \leq i < n$, $\phi_1 \in L(s_i)$. By inductive hypothesis, it follows that $r^n \nvDash \phi_2$ or it exists $0 \leq i < n$ such that $r^i \nvDash \phi_1$. Thus, by (5.2.2i) in Definition 5.2.2 it follows that $r \nvDash \phi_1 \,\mathcal{U}\, \phi_2$, and by (5.2.2e) $r \vDash \dot{\neg}(\phi_1 \,\mathcal{U}\, \phi_2)$.

Theorem 5.5.11. *$\Phi \subseteq \mathsf{csLTL}$ is unsatisfiable if and only if there exists a closed systematic tableau for Φ.*

Proof.

$\Rightarrow$ Suppose that it does not exist a closed tableau for Φ, then the systematic tableau $\mathcal{T}_\Phi$ would be open. Let b be an open branch of $\mathcal{T}_\Phi$ and S its stages. By Lemma 5.5.12, $\mathsf{stores}(\mathsf{path}(S))$ is a model for Φ, thus Φ is satisfiable.

$\Leftarrow$ Let $\mathcal{T}_\Phi$ be the closed systematic tableau for Φ. This means that the set of formulas labeling each leaf is unsatisfiable. By the algorithm in Definition 5.5.9 and by Lemma 5.5.1, it follows that every node in $\mathcal{T}_\Phi$ is labeled with an unsatisfiable set of formulas. Thus, Φ is unsatisfiable.

6 Implementation

In this chapter we describe the architecture of the concept prototype that we have developed, which is available online at URL `http://safe-tools.dsic.upv.es/tadi/`. This prototype aims to implement the whole semantics and abstract diagnosis framework for *tccp* that was described in this thesis. Particular emphasis has been posed on a modular and non-replicated development in order to gain maintainability, scalability and extensibility.

The prototype is written in Haskell (see [72, 98] for further details) and developed in the Glasgow Haskell Compiler (GHC) [99].

Figure 6.1 illustrates the prototype architecture. In the picture, the white modules are not implemented yet. Thus, our current prototype implements essentially the approach described in Chapter 5. We preferred to start with that part due to its previously mentioned advantages w.r.t. the abstract traces approach of Chapter 4.

The tool suite is composed of two main parts:

- the parser suite (written with `Alex` and `Happy`) and
- the abstract semantics suite.

In the following, we describe in more detail all the modules of our prototype.

6.1 Parser Suite description

Currently, our parser suite consists of about 7500 lines of code and it has two main parts:

- the *tccp*($\mathbf{C}$) parser
- the csLTL($\mathbf{C}$) parser

Both parsers are defined parametrically over a constraint system $\mathbf{C}$. Thus, their implementation, in order to parse the constraints, relies on a shared constraint parser module. The *constraint parser* takes a syntactic constraint which is an element of a given constraint system $\mathbf{C}$ and builds a semantic constraint that can be used by the other modules of the suite. At the moment, we have implemented the parser for a constraint system that supports different constraints:

- linear disequalities over natural numbers (Constraint System 1.4.3),
- Herbrand constraints (Constraint System 1.4.2) and
- streams ([43]).

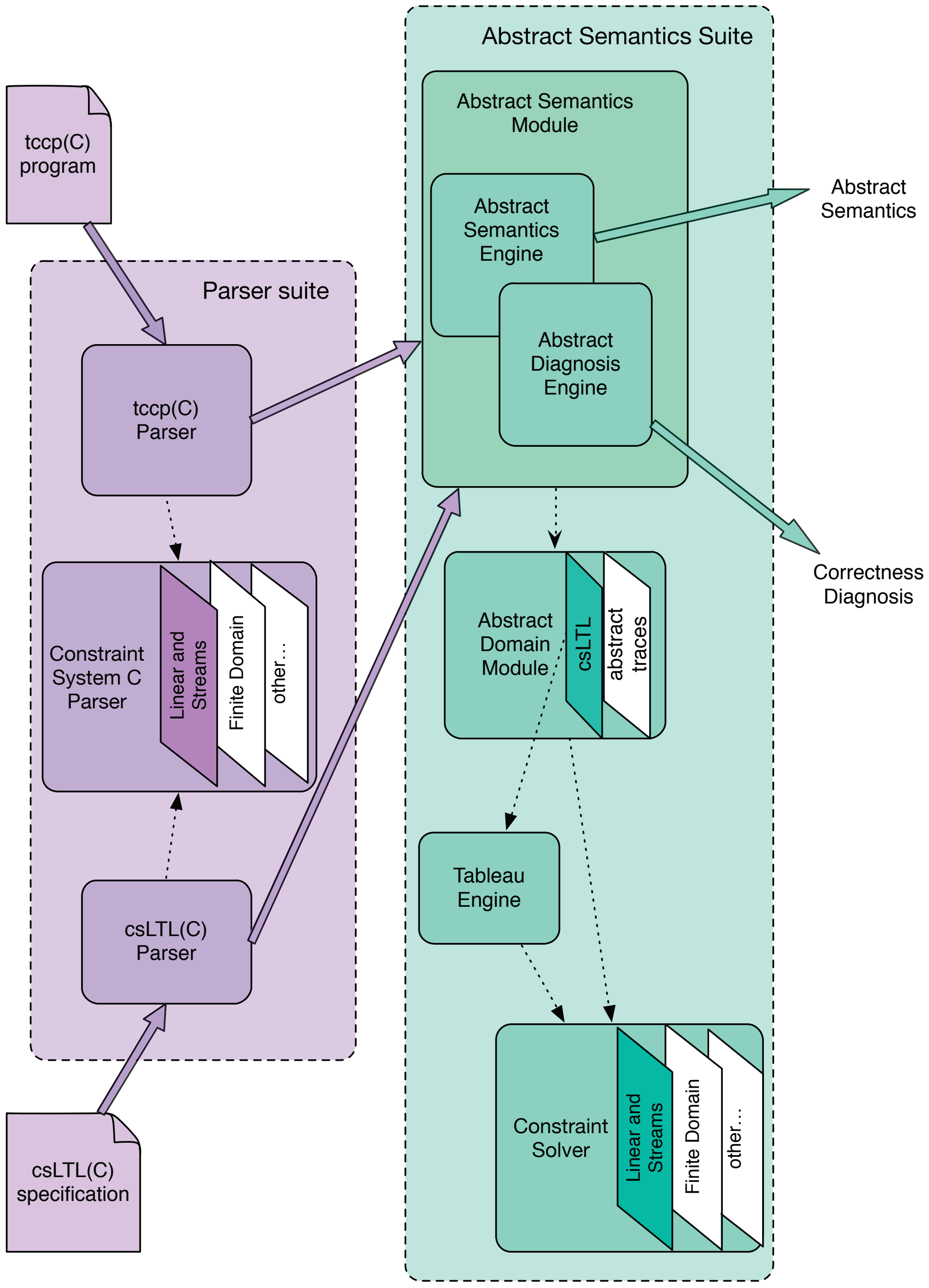


Figure 6.1: Prototype Architecture Diagram

```
⟨Constraint⟩   |    ⟨Constraint⟩ | ⟨Constraint⟩
               |    ⟨Constraint⟩ & ⟨Constraint⟩
               |    ! ⟨Constraint3⟩
               |    ⟨NonVarId⟩ ( ⟨AExprList⟩ )
               |    ⟨NonVarId⟩
               |    ⟨Expr⟩ = ⟨Expr⟩
               |    ⟨AExpr⟩ != ⟨AExpr⟩
               |    ⟨AExpr⟩ < ⟨AExpr⟩
               |    ⟨AExpr⟩ <= ⟨AExpr⟩
               |    ⟨AExpr⟩ > ⟨AExpr⟩
               |    ⟨AExpr⟩ >= ⟨AExpr⟩
               |    ⟨VarId⟩ :=: ⟨AExpr⟩

      ⟨Expr⟩  ::=   ⟨Stream⟩
               |    ⟨AExpr⟩

    ⟨Stream⟩  ::=   [ ⟨AExpr⟩ | ⟨VarId⟩ ]
```

Figure 6.2: BNF grammar for the constraint system

The concrete syntax of the constraints accepted by the parser is illustrated in Figure 6.2. Here, *VarId* is a variable identifier, *NonVarId* is a generic identifier (not for variables) and *AExpr* represents an arithmetic expression.

The *tccp parser* parses a *tccp* program defined over a given constraint system **C**. The concrete syntax of *tccp* accepted by the parser can be described by means of the BNF grammars illustrated in Figure 6.3. In this figure, the nonterminal symbol *Constraint* represents the syntax of the constraints of the underlying constraint system. Furthermore, *VarId* is a variable identifier, *VarIdList* is a list of variable identifiers, *ProcId* is a procedure name identifier and *AExprList* represents a list of arithmetic expression. As it can be noticed, we have added some syntactic sugar to write `ask(c)^n -> A`, instead of writing n nested agents which denote the repetition of the check ask(c) for n consecutive time units.

For example, the microwave agent of Example 3.1.24 can be written as:

```
microwave (Door, Button, Error) :-
    hid D,B,E (
    tell ( Error = [ _ | E ] )
    ||
    tell ( Door = [ _ | D ] )
    ||
    tell ( Button = [ _ | B ] )
    ||
    now ( Door = [ open | D ] & Button = [pushed | B] )
        then hid E1  tell(E = [yes | E1]) || hid B1  tell(B = [off | B1])
        else hid E1  tell(E = [no | E1])
    ||
    microwave (D,B,E) ).
```

$$
\begin{array}{rcl}
\langle \mathit{Program} \rangle & ::= & \{\ \langle \mathit{DeclList} \rangle\ \}\ \texttt{.}\ \langle \mathit{Agent} \rangle \\
\\
\langle \mathit{DeclList} \rangle & ::= & \epsilon \\
 & | & \langle \mathit{Decl} \rangle\ \texttt{.}\ \langle \mathit{DeclList} \rangle \\
\\
\langle \mathit{Decl} \rangle & ::= & \langle \mathit{ProcId} \rangle\ \texttt{(}\ \langle \mathit{VarIdList} \rangle\ \texttt{)}\ \texttt{:-}\ \langle \mathit{Agent} \rangle \\
 & | & \langle \mathit{ProcId} \rangle\ \texttt{:-}\ \langle \mathit{Agent} \rangle \\
\\
\langle \mathit{Agent} \rangle & ::= & \texttt{(}\ \langle \mathit{Agent} \rangle\ \texttt{)} \\
 & | & \texttt{skip} \\
 & | & \texttt{tell}\ \texttt{(}\ \langle \mathit{Constraint} \rangle\ \texttt{)} \\
 & | & \langle \mathit{Agent} \rangle \parallel \langle \mathit{Agent} \rangle \\
 & | & \langle \mathit{ProcId} \rangle\ \texttt{(}\ \langle \mathit{AExprList} \rangle\ \texttt{)} \\
 & | & \langle \mathit{ProcId} \rangle \\
 & | & \langle \mathit{ListAsk} \rangle \\
 & | & \langle \mathit{TkHiding} \rangle\ \langle \mathit{VarIdList} \rangle\ \langle \mathit{Agent} \rangle \\
 & | & \texttt{now}\ \texttt{(}\ \langle \mathit{Constraint} \rangle\ \texttt{)}\ \texttt{then}\ \langle \mathit{Agent} \rangle\ \texttt{else}\ \langle \mathit{Agent} \rangle \\
\\
\langle \mathit{Ask} \rangle & ::= & \texttt{ask}\ \texttt{(}\ \langle \mathit{Constraint} \rangle\ \texttt{)}\ \langle \mathit{OptInt} \rangle\ \texttt{->}\ \langle \mathit{Agent} \rangle \\
\\
\langle \mathit{ListAsk} \rangle & ::= & \langle \mathit{Ask} \rangle \\
 & | & \langle \mathit{Ask} \rangle\ \texttt{+}\ \langle \mathit{ListAsk} \rangle \\
\\
\langle \mathit{OptInt} \rangle & ::= & \epsilon \\
 & | & \texttt{\^{}}\ \langle \mathit{Integer} \rangle \\
\\
\langle \mathit{TkHiding} \rangle & ::= & \texttt{Hid} \mid \texttt{hid} \mid \texttt{hiding} \mid \texttt{local} \mid \texttt{exists}
\end{array}
$$

Figure 6.3: BNF grammar for *tccp* programs

The csLTL *parser* parses a csLTL formula which is parametric w.r.t. a given constraint system **C**. Also the nonstandard temporal LTL operators (such as Weak Until and Release) are available. The concrete syntax of csLTL accepted by the parser is illustrated in Figure 6.4. As before, the symbol *Constraint* represents the syntax of the constraints of the underlying constraint system and the symbol *VarIdList* represents a list of variable identifiers. Also in this case, we have added some syntactic sugar to write formulas of the form "`()^n`" to denote the repetition of the next operator n times. For example, suppose that we want to check for the microwave program the csLTL formula $\Box(\mathit{Button}\dot{=}\mathit{pushed} \dot{\rightarrow} (\bigcirc^5(\mathit{Door}\dot{=}\mathit{open}) \dot{\vee} \mathit{Error}\dot{=}\mathit{yes}))$. That specification can be written as:

```
microwave (Door, Button, Error) :
  [] (Button :=: pushed -> (next ^ 5 (Door :=: open) or Error :=: yes))
```

6.2 Abstract Semantics Suite description

The core of our prototype is the general abstract semantics module that is used to compute over abstract domains. It is composed by a *domain independent* part, which is a collection of functions implementing the abstract semantics evaluation functions for *tccp*. This part is parametric to an abstract domain module which in turn relies on a constraint solver.

The abstract domain module contains the structure of the operations and functions related to a general abstract domain. The specific definitions of the functions for each abstract domain instance are implemented in the submodules. The same scheme is applied to the constraint solver. Hence, each submodule implements the specific primitive functions of a given constraint system.

The abstract semantics suite counts about 3000 lines of code. Let us present in more details the principal components of this module.

6.2.1 Abstract Semantics Engine

The abstract semantics engine is composed by a collection of functions implementing the abstract semantics evaluation functions for *tccp*: $\mathcal{A}^\alpha$, $\mathcal{D}^\alpha$, $\mathcal{F}^\alpha$. This engine is parametric to an abstract domain module and a constraint solver.

This part is a large set of class declarations layered into different levels of abstraction (e.g. interpretation level, single interpretation binding level, domain level). The code that defines the default methods relies on lower level functions which are defined in the submodules of the abstract domain module. For instance, the agent evaluation function $\mathcal{A}^\alpha$ is defined at the domain independent interpretation level and its definition depends on the abstract domain and constraint system specific primitives to evaluate the agent semantics.

6.2.2 Abstract Diagnosis Engine

This part implements the abstract diagnosis functionalities described in Section 4.1 and Section 5.1. In particular, it implements the detection of abstract incorrect rules and the generation of a testimony of abstract incorrectness.

```
         ⟨CsLTL⟩  ::=  ⟨Constraint⟩
                   |   ⟨TkTrue⟩
                   |   ⟨TkFalse⟩
                   |   ⟨TkNot⟩ ⟨CsLTL⟩
                   |   ⟨CsLTL⟩ ⟨TkOr⟩ ⟨CsLTL⟩
                   |   ⟨CsLTL⟩ ⟨TkAnd⟩ ⟨CsLTL⟩
                   |   ⟨CsLTL⟩ ⟨TkImplies⟩ ⟨CsLTL⟩
                   |   ⟨CsLTL⟩ ⟨TkIsImpliedBy⟩ ⟨CsLTL⟩
                   |   ⟨CsLTL⟩ ⟨TkIfAndOnlyIf⟩ ⟨CsLTL⟩
                   |   ⟨TkExists⟩ ⟨VarIdList⟩ ⟨CsLTL⟩
                   |   ⟨CsLTL⟩ ⟨TkUntil⟩ ⟨CsLTL⟩
                   |   ⟨CsLTL⟩ ⟨TkWeakUntil⟩ ⟨CsLTL⟩
                   |   ⟨CsLTL⟩ ⟨TkRelease⟩ ⟨CsLTL⟩
                   |   ⟨TkEventually⟩ ⟨CsLTL⟩
                   |   ⟨TkAlways⟩ ⟨CsLTL⟩
                   |   ⟨TkNext⟩ ⟨OptInt⟩ ⟨CsLTL⟩

        ⟨OptInt⟩  ::=  ϵ
                   |   ^ ⟨Integer⟩

        ⟨TkTrue⟩  ::=  True | TRUE
       ⟨TkFalse⟩  ::=  False | FALSE
         ⟨TkNot⟩  ::=  !! | not !! | Not !! | NOT
          ⟨TkOr⟩  ::=  || | or | Or | OR
         ⟨TkAnd⟩  ::=  /\ | and | And | AND
     ⟨TkImplies⟩  ::=  -> | implies | Implies | IMPLIES
 ⟨TkIsImpliedBy⟩  ::=  <- | isimpliedby | IsImpliedBy | ISIMPLIEDBY
 ⟨TkIfAndOnlyIf⟩  ::=  <-> | iff | Iff | IFF
      ⟨TkExists⟩  ::=  exists | Exists | EXISTS
       ⟨TkUntil⟩  ::=  until | Until | UNTIL
   ⟨TkWeakUntil⟩  ::=  weakuntil | WeakUntil | WEAKUNTIL
     ⟨TkRelease⟩  ::=  release | Release | RELEASE
  ⟨TkEventually⟩  ::=  <> | eventually | Eventually | EVENTUALLY
      ⟨TkAlways⟩  ::=  [] | always | Always | ALWAYS
        ⟨TkNext⟩  ::=  () | next | Next | NEXT
```

Figure 6.4: BNF grammar for csLTL formulas

Most of the methods are defined at the domain independent interpretation level. For instance, the detection of abstract incorrect rules, which has as input a *tccp* set of declarations D and an abstractly domain instance α (with its associated parser), depends on the domain specific primitives for parsing a user defined specification, computing the immediate consequences of the specification w.r.t. the input program ($\mathcal{D}^{\alpha}[\![D]\!]_{\mathcal{S}^{\alpha}}$), and comparing the result with the specification.

As illustrated in Figure 6.1 this engine, as well as the abstract semantics one, is parametric to an abstract domain module and a constraint solver.

6.2.3 Abstract Domain Module

The abstract domain module implements the primitive operations of an abstract domain: equivalence, comparison, meet, join, renaming application, *etc.*

The main module contains just the general scheme, while the specific definitions of the operations are delegated to the submodules that implement the specific abstract domains of interest. For each abstract domain submodule it is necessary to define the corresponding parser which is used by the abstract diagnosis engine to acquire the program abstract specification.

Currently, we have provided a complete implementation of the csLTL domain (see Section 5.2). This abstract domain module counts about 1200 lines of code.

Tableau Engine

In order to implement the comparison in the domain of csLTL formulas, we need a way to decide if a formula ϕ logically implies another formula ψ (i.e., $\phi \dot{\rightarrow} \psi$). Our solution to this problem is to implement the tableau algorithm of Definition 5.5.9.

Our tableau engine is completely parametric w.r.t. an underlying constraint solver that is used to detect when a node is inconsistent (Definition 5.5.6) and to add new constraint formulas to a node through the merge operator $\otimes$.

The tableau engine consists of about 700 lines of Haskell code.

6.2.4 Constraint Solver

The Abstract Semantics Engine and the Abstract Diagnosis one use a common module that implements the constraint solver of the underlying constraint system $\mathbf{C}$. This module implements the basic operations of a constraint system: entailment relation, comparison, merge, join, cylindrification, *etc.*.

At the moment, we have already implemented a constraint solver for linear disequalities constraints, Herbrand constraints and streams. This solver consists of about 700 lines of code.

6.3 TADi: a Temporal Abstract Diagnosis Tool

TADi is a tool that implements the abstract diagnosis of *tccp* programs as described in Chapter 5. The underlying constraint system is formed by linear disequalities constraints, Herbrand constraints and streams.

Our online prototype is available online at URL `http://safe-tools.dsic.upv.es/tadi/`.

Figure 6.5 shows a screenshot of the TADi web interface. TADi expects a *tccp* program to be verified against a csLTL specification . It is possible to choose one of the predefined *tccp* programs by using the pulldown menu, to directly write a *tccp* program in the text area, or to modify the predefined ones. The predefined specification can be loaded for the selected *tccp* program or chosen by using the pulldown menu. Otherwise it is possible to write directly in the text area a desired specification or to modify the predefined ones. To check if the program meets the specification it is sufficient to press the Verify! button.

In the current version, TADi shows as result:

- The message "the process p is correct w.r.t. the specification" if the process respects the given specification.
- The message "the process p is not correct w.r.t. the specification on testimony φ", otherwise. In this message, $\varphi \in$ csLTL is the explicit testimony of the abstract incorrectness of p (see Proposition 5.5.14).

6.4 Future Work

In the future, we plan to implement other abstract domain modules, starting from the abstract conditional traces domain that was defined in Section 4.2. Furthermore, we want to implement other constraint systems such as the finite domain constraint system (see Example 4.2.1) or the sign domain constraint system (see Example 4.2.2).

We also plan to instantiate the abstract semantics module in order to obtain a semantic analysis tool. This tool will compute $\mathcal{D}^{\alpha}$ until a fixpoint is reached, thus it is terminating for instances over noetherian domains (e.g. $depth(k)$).

Additionally, we plan to extend it with narrowing and widening operators, two typical constructions of the abstract interpretation setting, which are mainly used to achieve termination for non-noetherian domains, but are also frequently used to speedup convergence for "big" noetherian domains.

Currently, the implementation of the constraint solver is pretty rough. Nevertheless, the results we obtained until now are quite interesting. We believe that after a refectory of this module we could get important improvements in terms of efficiency.

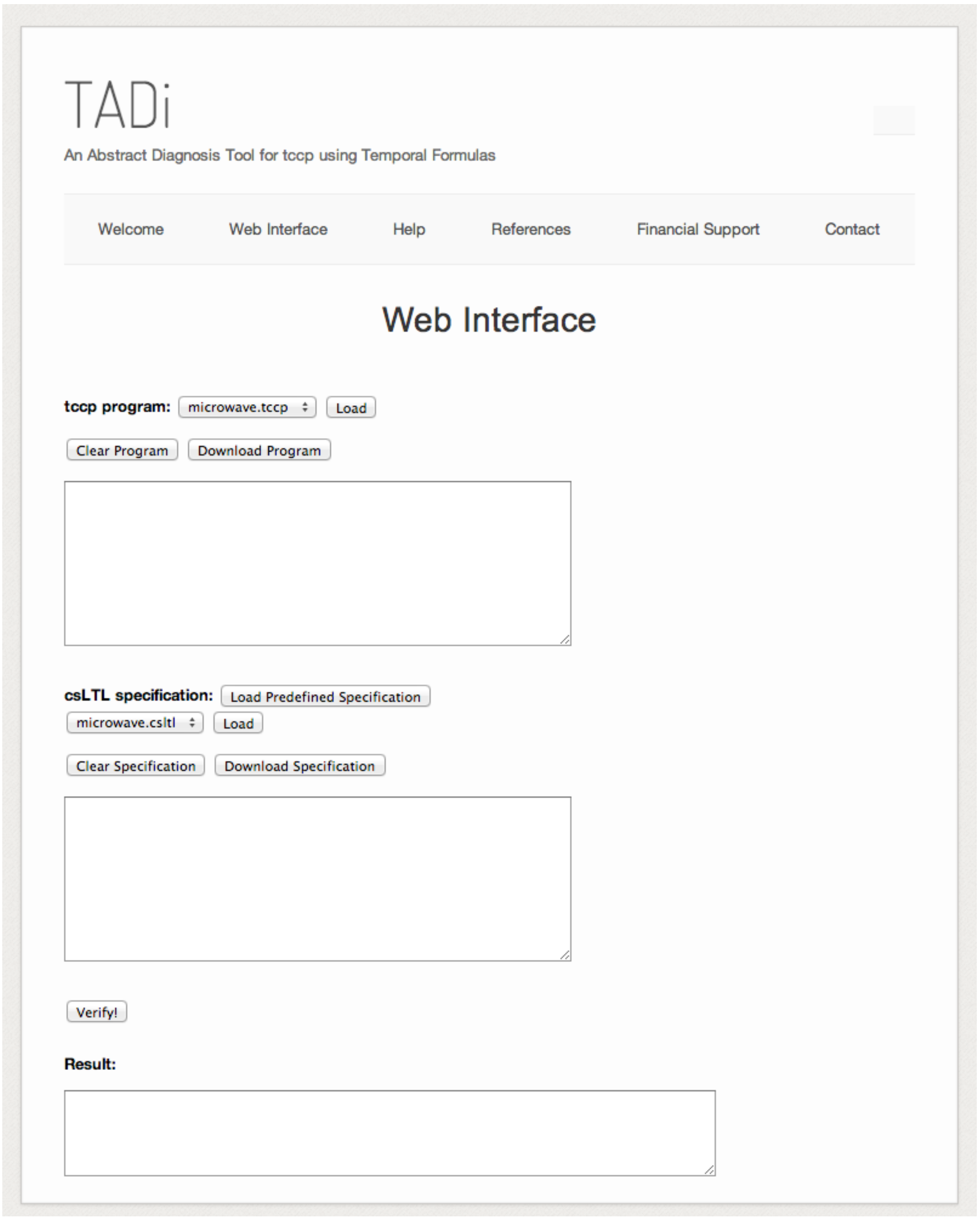


Figure 6.5: TADi web interface

Conclusions

In this thesis we have proposed an abstract interpretation framework for *tccp* with the aim of formally debugging and verifying concurrent and reactive systems.

This framework relies on the denotational semantics defined in Chapter 3 which models exactly the small-step behavior of *tccp*. We have shown that this semantics also fulfills some desired requirements that make it suitable to be used in the development of efficient semantic-based program manipulation techniques: goal-independency, condensed denotation, bottom-up definition and compositionality. To our knowledge, our proposal is the first fully abstract compositional semantics for a non-deterministic language in the *ccp* family which covers the whole language (including non-monotonic behaviors and infinite computations).

Additionally, we have defined a big-step semantics for *tccp* (as an abstraction of the small-step one) and we have shown that it is essentially equivalent to the original input-output semantics of the language [43]. Moreover, we have proven that it is not possible to have an enough precise and still correct input-output fixpoint semantics which is defined only on the information provided by the input/output pairs. This is due to the loss of synchronization between concurrent processes that happens when the information about the evolution of the store at each time instant is approximated by input/output pairs.

By using our small-step semantics we have defined two different abstract diagnosis instances which are able to formally debug and verify *tccp* programs:

Abstract diagnosis based on constraint system abstractions. In Chapter 4 we have defined a fully automatic debugging method for *tccp* programs based on abstract interpretation. This method is parametric w.r.t. an abstract domain (which models the properties of interest) and a Galois Insertion between the concrete domain of conditional traces and the abstract one. We have instantiated the method with an abstract domain composed by compact abstract conditional traces. Therefore, in our proposal, specifications are compact conditional abstract traces which contain only approximated information (i.e., approximated constraints). We showed that this approach is able to deal with the full *tccp* language and can validate properties of both finite and infinite computations.

Abstract diagnosis based on temporal formulas. In Chapter 5 we have extended the abstract diagnosis approach of Chapter 4 in order to obtain more intuitive specifications. We have defined a new abstract diagnosis framework which is parametric just to an abstract domain and a concretization function. This method can be used in general when the abstract function cannot be defined, i.e., the best correct approximation does not exist. We have instantiated this general framework with a domain formed by linear temporal formulas with constraints. In this way, a specification for a *tccp* set of declarations is a temporal logic formula, as in model checking. In order to make this method automatic we have presented a decision method for the temporal logic used which is based on the systematic construction of a tableau.

The most interesting feature of abstract diagnosis is that it can be used with partial specifications and partial programs and works even with non-terminating programs. Moreover, it does not require the user to provide error symptoms in advance and it is able to detect and locate multiple errors simultaneously.

Our proposals, does not require to build any model of the target program, but in order to detect all errors a specification of the program has to be given. As usual when using abstract semantics, the obtained result guarantees correctness but completeness is lost. For that reason, the results obtained by using abstract diagnosis can be less precise w.r.t. the results obtained by using traditional techniques such as model checking. However, all "visible" errors are assured to be detected, which makes our proposal suitable to be used to find errors in critical systems.

We have implemented a first prototype of the semantics and abstract diagnosis framework for *tccp* and the results are very encouraging (Chapter 6). In the immediate future, we plan to test it with complex *tccp* programs and compare its results and performance with other tools.

As future work, we plan to investigate further on applications of our semantics to obtain novel analysis and verification methods.

Moreover, we plan to adapt the ideas presented in this thesis to define appropriate fully-abstract semantics for other concurrent languages of the *ccp* family, such as *ntcc*, *utcc* and *tcc*. Along these lines, we will be able to straightforwardly adapt the abstract diagnosis methodology to such languages. These adaptations of the semantics are not immediate, since these languages have significant differences w.r.t. *tccp*, but we are confident that the required effort will be reasonable.

Bibliography

[1] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation – 12th International Workshop, LOPSTR 2002, Revised Selected Papers*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16, Berlin, 2003. Springer-Verlag. I.3.3, 4

[2] M. Alpuente, M. Falaschi, and A. Villanueva. Symbolic model checking for timed concurrent constraint programs. In *Actas de las III Jornadas de Programación y Lenguajes (PROLE'03)*, pages 151–165, Alicante (Spain), 2003. I.2.1

[3] M. Alpuente, M. Falaschi, and A. Villanueva. A Symbolic Model Checker for tccp Programs. In *First International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2004), Revised Selected Papers*, volume 3475 of *Lecture Notes in Computer Science*, pages 45–56. Springer-Verlag, 2005. 5, 5.6

[4] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. A Semantic Framework for the Abstract Model Checking of tccp Programs. *Theoretical Computer Science*, 346(1):58–95, 2005. I.2.1, 4.2.3, 4.5, 5, 5.6

[5] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. Abstract Model Checking of tccp programs. *Electronic Notes in Theoretical Computer Science*, 112:19–36, 2005. I.2.1

[6] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. Verifying Real-Time Properties of tccp Programs. *Journal of Universal Computer Science*, 12(11):1551–1573, 2006. 2.3, 5.3.4, 5.4

[7] A. Aristizábal, F. Bonchi, C. Palamidessi, L. F. Pino, and F. D. Valencia. Deriving Labels and Bisimilarity for Concurrent Constraint Programming. In *14th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2011)*, volume 6604 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011. 1.4

[8] G. Bacci and M. Comini. Abstract Diagnosis of First Order Functional Logic Programs. In M. Alpuente, editor, *Logic-based Program Synthesis and Transformation, 20th International Symposium*, volume 6564 of *Lecture Notes in Computer Science*, pages 215–233, Berlin, 2011. Springer-Verlag. I.3.3, 4

[9] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985. (document)

[10] G. Berry. Preemption in concurrent systems. In R. K. Shyamasundar, editor, *FSTTCS*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93. Springer, 1993. I.1

[11] G. Berry. The foundations of Esterel. In G. D. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction*, pages 425–454. The MIT Press, 2000. I.1

[12] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992. 3.3

[13] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:117–148, 2003. 5

[14] G. Birkhoff. *Lattice Theory*, volume 25 of *AMS Colloquium Publication*. American Mathematical Society, 1967. 1, 1.3.1

[15] G. Birkhoff and S. MacLane. *A Survey of Modern Algebra*, volume 25. MacMillan, 1965. Third edition. 1

[16] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. I.2

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992. I.2, 5

[18] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM. I.1

[19] K.L. Clark. *Predicate Logic as a Computational Formalism*. Research monograph / Department of Computing, Imperial College of Science and Technology, University of London. University of London, 1980. 1.4

[20] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag. I.2, 5

[21] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: A practical approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 117–126, New York, NY, USA, 1983. ACM. I.2

[22] E. M. Clarke, O Grumberg, and Long D. E. Model checking. In *NATO ASI DPD*, pages 305–349, 1996. (document)

[23] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Conference Record of the 19th ACM Symp. on Principles of Programming Languages (POPL'92)*, pages 343–354, New York, NY, USA, 1992. ACM Press. 5

[24] M. Comini. *An Abstract Interpretation Framework for Semantics and Diagnosis of Logic Programs*. PhD thesis, Dipartimento di Informatica, Universitá di Pisa, Pisa, Italy, 1998. 1

[25] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999. I.2, I.3.3, I.4, 4, 4.1

[26] M. Comini, L. Titolo, and A. Villanueva. Abstract Diagnosis for Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming*, 11(4-5):487–502, 2011. I.6

[27] M. Comini, L. Titolo, and A. Villanueva. A Condensed Goal-Independent Bottom-Up Fixpoint Modeling the Behavior of tccp. Submitted for publication., 2013. I.6

[28] M. Comini, L. Titolo, and A. Villanueva. Towards an Effective Decision Procedure for LTL formulas with Constraints. In *23rd Workshop on Logic-based methods in Programming Environments (WLPE 2013)*, volume abs/1308.2055, 2013. I.6

[29] M. Comini, L. Titolo, and A. Villanueva. Abstract Diagnosis for tccp using a Linear Temporal Logic. *Theory and Practice of Logic Programming*, 14(4-5), 2014. To appear. I.6

[30] T. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988. I.2

[31] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Los Angeles, California, January 17–19*, pages 238–252, New York, NY, USA, 1977. ACM Press. I.2, I.3, I.4, 1.3

[32] P. Cousot and R. Cousot. A constructive characterization of the lattices of all retracts, pre-closure, quasi-closure and closure operators on a complete lattice. *Portugaliæ Mathematica*, 38(2):185–198, 1979. 1.3.1

[33] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, January 29–31*, pages 269–282, New York, NY, USA, 1979. ACM Press. I.2, I.3, 1.3, 1.3.1, 1.3.2, 1.3.2, 3.2

[34] P. Cousot and R. Cousot. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming*, 13(2 & 3):103–179, 1992. 1.3.2

[35] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–549, 1992. 5.1

[36] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, Proceedings PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295, Berlin, 1992. Springer-Verlag. 1.3.3

[37] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER

analysis of functional languages). In *Proceedings of the IEEE International Conference on Computer Languages (ICCL'94)*, pages 95–112, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. 1.3.4

[38] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of Fifth ACM Symp. Principles of Programming Languages*, pages 84–96, 1978. 5.1, 5.1

[39] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997. 1.3.4

[40] D. R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, 1996. 5

[41] F. S. de Boer, A. di Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151:37–78, 1995. 1.4, 1.4, 2.3, 3, 3.1, 3.1.24, 3.3

[42] F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving Concurrent Constraint Programs Correct. *ACM Trans. Program. Lang. Syst.*, 19(5):685–725, 1997. 1.4, 1.5, 3.1, 3.1.2, 3.1.18, 3.3

[43] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000. (document), I.1, I.4, I.5, 1.4, 1.4.1, 2, 2.2, 2.3, 2.3, 3, 3.1.24, 3.2, 3.2.1, 3.2.2, 3.2.2, 3.4, 3.A.2, 4.2.3, 4.4.2, 4.4.3, 6.1, 6.4

[44] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Temporal Logic for Reasoning about Timed Concurrent Constraint Programs. In *TIME '01: Proceedings of the Eighth International Symposium on Temporal Representation and Reasoning (TIME'01)*, page 227, Washington, DC, USA, 2001. IEEE Computer Society. I.2.1, 5, 5.2

[45] F. S. de Boer, M. Gabbrielli, and M. C. Meo. Proving correctness of Timed Concurrent Constraint Programs. *CoRR*, cs.LO/0208042, 2002. I.2.1, 5, 5.2, 5.6

[46] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. M. M. Rutten. The Failure of Failures in a Paradigm for Asynchronous Communication. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR*, volume 527 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 1991. 1.4, 1.4

[47] F. S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T. S. E. Maibaum, editors, *Proceedings of TAPSOFT'91*, volume 493 of *Lecture Notes in Computer Science*, pages 296–319, Berlin, 1991. Springer-Verlag. 3, 3.2

[48] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag. I.2

[49] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. *Theoretical Computer Science*, 183:281–315, 1997. 3, 3.1, 3.3

[50] M. Falaschi, C. Olarte, and C. Palamidessi. A Framework for Abstract Interpretation of Timed Concurrent Constraint Programs. In A. Porto and F. J. López-Fraguas, editors, *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 207–218, New York, NY, USA, 2009. Acm. I.2.1, 3.1, 3.3, 4.5, 4.6

[51] M. Falaschi, C. Olarte, C. Palamidessi, and F. D. Valencia. Declarative Diagnosis of Temporal Concurrent Constraint Programs. In V. Dahl and I. Niemelä, editors, *Logic Programming, 23rd International Conference, ICLP 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 271–285. Springer-Verlag, 2007. I.2.1, I.3.3, 3.1, 3.3, 4, 4.4, 4.4.3, 4.5, 4.6

[52] M. Falaschi, A. Policriti, and A. Villanueva. Modeling concurrent systems specified in a temporal concurrent constraint language-I. *Electronic Notes in Theoretical Computer Science*, 48:197–210, 2001. 5.6

[53] M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. *Theory and Practice of Logic Programming*, 6(3):265–300, 2006. I.2.1, 2.3, 5, 5.6

[54] G. Ferrand. Error Diagnosis in Logic Programming, an Adaptation of E. Y. Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987. I.2

[55] G. Filè, R. Giacobazzi, and F. Ranzato. A Unifying View on Abstract Domain Design. *ACM Computing Surveys*, 28(2):333–336, 1996. 1.3.3

[56] R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Symp. in Applied Mathematics of the AMS*, pages 19–32, 1967. I.2

[57] J. Gaintzarain. *Invariant-Free Deduction Systems for Temporal Logic.* PhD thesis, Department of Computer Languages and Systems, University of the Basque Country, 2012. 5.5, 5.5.1, 5.5.1, 5.5.2, 5.5.2, 5.5.7, 5.5.8, 5.5.3, 5.5.3

[58] J. Gaintzarain, M. Hermo, P. Lucio, and M. Navarro. Systematic semantic tableaux for PLTL. *Electronic Notes in Theoretical Computer Science*, 206:59–73, 2008. I.4, 5, 5.5.1, 5.5.1, 5.5.2

[59] J. Gaintzarain, M. Hermo, P. Lucio, M. Navarro, and F. Orejas. A cut-free and invariant-free sequent calculus for PLTL. In J. Duparc and T. A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 481–495. Springer, 2007. 5.5.1

[60] J. Gaintzarain, M. Hermo, P. Lucio, M. Navarro, and F. Orejas. Dual Systems of Tableaux and Sequents for PLTL. *The Journal of Logic and Algebraic Programming*, 78(8):701–722, 2009. I.4, 5, 5.5, 5.5.1, 5.5.1, 5.5.2, 5.5.2, 5.5.7, 5.5.8, 5.5.3, 5.5.3, 5.5.5, 5.A.3, 5.A.3

[61] T. Gautier and P. Le Guernic. SIGNAL: A declarative language for synchronous programming of real-time systems. In G. Kahn, editor, *FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 257–277. Springer, 1987. I.1

[62] R. Giacobazzi and F. Ranzato. Functional dependencies and Moore-set completions of abstract interpretations and semantics. In J. W. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming (ILPS'95)*, pages 321–335, Cambridge, Mass., 1995. The MIT Press. 1.3.3

[63] R. Giacobazzi and F. Ranzato. Completeness in abstract interpretation: A domain perspective. In M. Johnson, editor, *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, Lecture Notes in Computer Science, Berlin, 1997. Springer-Verlag. 1.3.4

[64] R. Giacobazzi and F. Scozzari. Intuitionistic Implication in Abstract Interpretation. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of Ninth International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, volume 1292 of *Lecture Notes in Computer Science*, pages 175–189, Berlin, 1997. Springer-Verlag. 1.3.3

[65] V. Gupta, R. Jagadeesan, and V. A. Saraswat. Hybrid cc, hybrid automata and program verification. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 52–63. Springer, 1995. I.1

[66] V. Gupta, V. A. Saraswat, and P. Struss. Modeling a Photocopier Paper Path. In *Proceedings of the 2nd IJCAI Workshop on Engineering Problems for Qualitative Reasoning*, 1995. I.1

[67] L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras. Part I and II.* Elsevier Science Publishers, North Holland, 1971. 1.4

[68] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994. 5

[69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. I.2

[70] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. (document)

[71] G. J. Holzmann. On-The-Fly model checking. *ACM Comput. Surv.*, 28(4es):120, 1996. 5

[72] P. Hudak, J. Peterson, and J. Fasel. *A Gentle Introduction to Haskell - Version 98.* `http://www.haskell.org/tutorial/`, 1999. 6

[73] G. E. Hughes and M. J. Cresswell. *A New Introduction to Modal Logic.* Routledge, 1996. 1.5

[74] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992. 3.1

[75] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing computer software (2. ed.).* Van Nostrand Reinhold, 1993. I.2

[76] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987. I.2

[77] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, NewYork, 1974. 1

[78] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems - specification.* Springer, 1992. I.2.1, 1, 1.5, 5.2

[79] Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety.* Springer, 1995. (document), 2.3

[80] K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993. 3.1

[81] K. L. McMillan. *Symbolic Model Checking.* Kluwer, 1993. I.2

[82] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1980. (document)

[83] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992. I.1

[84] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41–77, 1992. I.1

[85] A. Mycroft. Completeness and predicate-based abstract interpretation. In *Proceedings of the ACM Symp. on Partial Evaluation and Program Manipulation (PEPM'93)*, pages 179–185, New York, NY, USA, 1993. ACM Press. 1.3.4

[86] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 1979. I.2

[87] M. Nielsen, C. Palamidessi, and F. D. Valencia. On the Expressive Power of Temporal Concurrent Constraint Programming Languages. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 156–167, New York, NY, USA, 2002. ACM Press. I.1, 3, 3.3, 4.2.1

[88] M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal Concurrent Constraint Programming: Denotation, Logic and Applications. *Nordic Journal of Computing*, 9(1):145–188, 2002. I.2.1, 1.5, 3.1, 3.1.2, 3.1.18, 3.3

[89] C. Olarte, C. Palamidessi, and F. D. Valencia. Universal timed concurrent constraint programming. In *ICLP'07: Proceedings of the 23rd international conference on Logic programming*, pages 464–465, Berlin, Heidelberg, 2007. Springer-Verlag. I.1

[90] C. Olarte and F. D. Valencia. Universal concurrent constraint programing: symbolic semantics and applications to security. In R. Wainwright and H. Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC08)*, pages 145–150. ACM, 2008. I.1, 3.1, 3.3

[91] O. Ore. Galois Connections. *Transactions of the American Mathematical Society*, 55:493–513, 1944. 1.3.2, 1.3.2

[92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *CADE*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992. I.2

[93] C. Palamidessi and F. Valencia. A temporal concurrent constraint programming calculus. In T. Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 2001. I.1, I.2.1, 5.6

[94] C. Palamidessi and F. D. Valencia. A Temporal Concurrent Constraint Programming Calculus. In *7th International Conference on Principles and Practice of Constraint Programming (CP'01)*, volume 2239 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 2001. 5, 5.2

[95] R. Patton. *Software Testing.* Sams, 2000. I.2

[96] L. C. Paulson. Natural Deduction as Higher-Order Resolution. *J. Log. Program.*, 3(3):237–258, 1986. I.2

[97] D. Peled. *Software Reliability Methods.* Texts in Computer Science. Springer, 2001. 1.5

[98] S. Peyton Jones. *Haskell 98 Language and Libraries - The Revised Report.* Cambridge University Press, Cambridge, UK, 2003. Available at `http://www.haskell.org/definition/`. 6

[99] S. L. Peyton Jones, C. Hall, K. Hammond, Jones Cordy, H. Kevin, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview, 1992. 6

[100] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982. I.2, 5

[101] U. S. Reddy and S. N. Kamin. On the power of abstract interpretation. In *Proceedings of the 1992 IEEE Internat. Conf. on Computer Languages* (*ICCL'92*), Los Alamitos, CA, USA, 1992. IEEE Computer Society Press. 1.3.4

[102] B. Russell. *The Principles of Mathematics.* Number vol. 1 in The Principles of Mathematics. University Press, 1903. 1.1.1

[103] V. A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989. I.1

[104] V. A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, Mass., 1993. (document), I.1, 1, 1.4, 2, 4

[105] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Programming in Timed Concurrent Constraint Languages. In B. Mayoh, E. Tyugu, and J. Penjaam, editors, *Constraint Programming: Proceedings 1993 NATO ASI, Berlin*, pages 361–410. Springer-Verlag, 1993. I.1

[106] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80. IEEE Computer Press, 1994. I.1, (document), 3.3

[107] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Default Timed Concurrent Constraint Programming. In R. K. Cytron and P. Lee, editors, *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 272–285, San Francisco, California, USA, 1995. ACM Press. 3.3

[108] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed Default Concurrent Constraint Programming. *Journal on Symbolic Computation*, 22(5/6):475–520, 1996. I.1, (document), 3.1, 3.3

[109] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1990. ACM. I.1, 1.4, 2

[110] V. A. Saraswat, M. Rinard, and P. Panangaden. The Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–352, New York, NY, USA, 1991. ACM. I.1, 1.4, 1.4, 2, 2.2, 3.3

[111] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999. 2.3

[112] R. C. Sekar, P. Mishra, and I. V. Ramakrishnan. On the power and limitation of strictness analysis. *Journal of the ACM*, 44(3):505–525, 1997. 1.3.4

[113] E. Y. Shapiro. Algorithmic Program Debugging. In *Proceedings of Ninth Annual ACM Symp. on Principles of Programming Languages*, pages 412–531, New York, NY, USA, 1982. ACM Press. I.2, 4, 4.1

[114] A. Tarski. A Lattice-theoretical Fixpoint Theorem and its Applications. *Pacific J. Math.*, 5:285–309, 1955. 1.2.1

[115] L. Titolo. An Abstract Interpretation Framework for Verification of Timed Concurrent Constraint Languages. *Theory and Practice of Logic Programming*, 13(4-5-Online-Supplement), 2013. I.6

[116] F. D. Valencia. Decidability of infinite-state timed CCP processes and first-order LTL. *Theoretical Computer Science*, 330(3):577–607, 2005. I.2.1, 1.5, 5, 5.2, 5.6, 5.7