An Abstract Interpretation Framework for Verification of Timed Concurrent Constraint Languages

LAURA TITOLO∗

Department of Mathematics and Computer Science - University of Udine
Via delle Scienze, 206 – 33100 Udine, Italy
(*e-mail:* `laura.titolo@uniud.it`)

submitted 9 April 2013; revised TBD; accepted TBD

Abstract

The Timed Concurrent Constrain programming (*tccp* in short) is a concurrent logic language obtained by extending the *cc* paradigm with the notion of time and a suitable mechanism to model time-outs and preemptions. Thanks to these features, *tccp* is a suitable language to model both concurrent and reactive systems.

The existing formal techniques for the verification of *tccp* are based on model checking. Given a program P and a temporal formula ϕ, model checking essentially works by browsing the structure of some form of model M (which represents the behavior of P) to check if ϕ is valid. This implies that the model M has to be built and, especially in presence of concurrency, this model is quite huge. This problem is well known in the literature and it is called state-explosion problem.

The core of my research is the definition of a semantic framework for *tccp* for the formal verification of complex concurrent and reactive systems by using abstract interpretation theory. This approach is based on the definition of a concrete semantics for *tccp* which is suitable to be approximated by using standard abstract interpretation results. In this way, it is possible to define debugging and verification tools for *tccp* without the burden of modeling the system first. Thus, abstract interpretation becomes a valid alternative to model checking, since it overcomes the state-explosion problem.

1 Introduction

Nowadays, time aspects are essential to an increasingly large number of applications such as the modeling, specification and verification of concurrent and reactive systems. A *concurrent system* contains different components that run in parallel and interact with each other. *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.

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. For example, systems that implement electronic financial transitions, electronic commerce, medical instruments, or air traffic control.

∗ This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN, ref. TIN 2010-21062-C02-0

Formal methods are a collection of notations and techniques for describing, verifying and analyzing systems. For critical systems it is necessary to use formal methods in order to be sure that the system behaves correctly. Also non-critical systems can be object of formal verification whenever possible.

Many formalisms have been developed to model concurrent systems. One of those formalisms is the *Concurrent Constraint* (*cc*) *paradigm* of Saraswat (Saraswat 1993). 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 check the absence of information, as well as its presence. The *Timed Concurrent Constrain programming* (de Boer et al. 2000) (*tccp* in short) is a concurrent logic language obtained by extending *ccp* with the notion of time and a suitable mechanism to model time-outs and preemptions. Thanks to these features, *tccp* is a suitable language to model both concurrent and reactive systems.

The existing formal techniques for the verification of *tccp* are based on model checking. Model checking (Clarke et al. 1996) 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.

The core of my research is the definition of a semantic framework for *tccp* based on abstract interpretation with the aim of formally verify complex concurrent and reactive systems. The first step towards this purpose is the definition of a concrete denotational semantics for *tccp* which is suitable to be used in the abstract interpretation framework. Then, this concrete semantics is approximated in order to define debugging and verification tools for *tccp*, which are correct by construction, by using standard results of abstract interpretation theory. The main advantage of this approach is that, contrary to model checking, it is not necessary to build the model of the system (which can be huge).

2 Background and overview of the existing literature

In this section, I present an overview of the state of the art regarding the modeling and the formal verification of concurrent and reactive systems in the context of the concurrent constraint paradigm.

2.1 Modeling concurrent and reactive systems

Concurrent systems consist of multiple *agents* (also called *processes*) that interact among each other. There are many models for concurrent systems. Some examples are the process calculi CCS (Milner 1980), CSP (Hoare 1978) or the ACP (Bergstra and Klop 1985).

The model considered in this thesis is the *Concurrent Constraint paradigm* (*cc* paradigm or *ccp*) defined in (Saraswat 1989; Saraswat 1993; Saraswat and Rinard 1990) as a simple and powerful model of concurrent computation. In this computational model, the notion of *store-as-valuation* from von Neumann is replaced with the notion of *store-as-constraint.* Thus, the *cc* paradigm is parametric w.r.t. a *constraint system* (see (Saraswat et al. 1991)) where, instead of knowing the specific value of a variable, just partial information is available. In this formalism, the agents exchange information through a global constraint store that is common to all agents. Essentially, 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 (Saraswat et al. 1993; Saraswat et al. 1994) was inspired by synchronous languages such as ESTEREL (Berry 2000), Lustre (Caspi et al. 1987) or SIGNAL (Gautier and Le Guernic 1987). *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 (Saraswat et al. 1994), the essence of the time-out and the preemptions mechanisms is in the ability to detect the *absence* of an event, as well as its presence.

In the following years, some extensions of *tcc* were introduced to improve the expressivity of the language and to model more complex systems. In (Saraswat et al. 1996), the *Timed Default Concurrent Constraint programming* is defined 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. The notion of non-determinism was introduced into the *tcc* model a few years later in (Palamidessi and Valencia 2001) by defining the *ntcc* language. Finally, in 2007, Olarte, Palamidessi and Valencia introduced the Universal Timed Concurrent Constraint language (*utcc*) (Olarte et al. 2007; Olarte and Valencia 2008) with the aim of extending *tcc* with the notion of mobility in the sense of Milners π-calculus (Milner et al. 1992a; Milner et al. 1992b).

As an alternative to *tcc*, 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*) (de Boer et al. 2000). *tccp* is a Turing powerful non-deterministic language that interprets the parallel composition in terms of maximal parallelism. The notion of time is introduced by means of a global discrete clock. 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). Furthermore, it introduces some constructs that check for the absence of information in the constraint store, allowing one to implement behaviors typical of reactive systems, such as *time-out* and *weak preemption*. Thanks to its features, *tccp* is suitable to model large and complex concurrent timed systems. For this reason, we choose to define an abstract interpretation framework for semantics and verification of *tccp* programs.

2.2 Modeling the behavior in timed concurrent constraint languages

In order to apply efficiently semantics-based verification techniques we need a semantics that is fully-abstract w.r.t. the small-step behavior of *tccp*, compositional, bottom-up, goal-independent and condensed. The semantics already proposed in the literature for *tccp* and for the other similar languages of the Concurrent Constraint paradigm do not meet all these good properties. Let us present an overview of these semantics.

In (de Boer et al. 1995), the difficulties for handling nondeterminism and infinite behavior in the *ccp* paradigm were investigated. The authors showed that the presence of nondeterminism and synchronization require 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. In most of the semantics of the *cc* paradigm the solution to all these difficulties has been traditionally based on the introduction of restrictions on the language for the definition of the semantics. These restrictions usually regards choice agents, hiding operators (which introduce local variables) and non-monotonic operators. Non-monotonic operators are used to detect the absence of information and are essential to model specific behaviors of reactive systems, such as time-out or preemption. In fact, they give rise to non-monotonic behaviors, which are complex to be modeled in a compositional semantics.

The semantics for *tcc* (Saraswat et al. 1994)), *ntcc* (Palamidessi and Valencia 2001) and *utcc* (Falaschi et al. 2009; Olarte and Valencia 2008) avoid the non-monotonic behavior in order to have fully abstraction and compositionality. The considered fragment is called locally-independent and avoids the interaction between local and non-monotonic operators. These semantics model the small-step behavior (i.e., how the state evolves at each time instant) under the assumption of absence of stimuli from the external environment, which is modeled as an integrated part of the program.

In (de Boer et al. 2000) the authors presented a denotational semantics fully-abstract w.r.t. the input-output behavior of *tccp* programs, under some assumptions on the underlying constraint system. However, this semantics only deals with finite computations and it does not consider infinite computations that are essential to describe the behavior of reactive systems.

To our knowledge, the only compositional semantics that is fully abstract w.r.t. the small-step behavior for a dialect of *ccp* with non-monotonic behavior was defined for the *Default tcc* language (Saraswat et al. 1999), and it makes use of default values (stable assumptions for the negative information) in order to model *strong preemption.* This also aids to overcome the problem of the non-monotonic behavior. However, this semantics is not condensed since it contains a lot of redundant information.

2.3 Formal verification of timed concurrent constraint languages

The verification of a system consists in checking its correctness w.r.t. a given intended behavior. Formal verification techniques are based on some mathematical theory and can assure that a program behaves as expected. Thus, they are suitable to verify critical systems, since they can assure the absence of errors.

The existing formal techniques for the verification of *tccp* are based on model checking (Falaschi and Villanueva 2006). 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: a system with n identical processes each of them having m states, in the worse case has m^n states. To mitigate this problem two different approaches were proposed: the abstract model checking in (Alpuente et al. 2005b; Alpuente et al. 2005a) that reduce the size of the initial model by means of an approximation, and the symbolic model checking in (Alpuente et al. 2003) that use a symbolic representation of the model. Although these methods enhance the applicability of model checking, the combinatory explosion of the states, especially for concurrent systems, is still an intrinsic problem.

A possible approach to avoid this issue is the use of approximation techniques, such as abstract interpretation, in order to define verification and diagnosis tools. Because of the approximation, these techniques are correct but in general not complete, since they can originate false positive. However, they are a good alternative to model checking, since they are in general more efficient and they can assure the absence of errors in a system w.r.t. the expected behavior.

In (Falaschi et al. 2007), a first approach to the declarative debugging of a *ccp* language is presented. Falaschi *et al.* introduce a semantic framework for *ntcc* and, by using standard abstract interpretation techniques, they define an abstract diagnosis method. This approach has some drawbacks. First of all, it does not cover the particular extra difficulty of modeling the semantics of non-monotonic operators. Furthermore, the authors approximate infinite sequences by cutting it at a given depth, thus they cannot verify with enough precision the infinite behaviors typical of reactive systems. In 2009 a similar approach was presented in (Falaschi et al. 2009) for *utcc*.

In (de Boer et al. 2001; de Boer et al. 2002), 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 in (Manna and Pnueli 1992) where logic propositions are replaced with constraints.

An analogous work was made for *ntcc*. In (Palamidessi and Valencia 2001; Nielsen et al. 2002) *ntcc* is equipped with a temporal logic, called CLTL (Constraint LTL), able to express program specifications. In (Valencia 2005), some decidability results for the verification of *ntcc* programs using CLTL specifications are presented. The author shows that for the locally-independent fragment of *ntcc*, it is possible to automatically verify a negation-free CLTL formula. The decidability of this verification follows from the monotonicity of the locally-independent fragment and the absence of recursion, but it can be noticed that not even this approach is able to deal with the whole language, in particular with non-monotonic operators.

3 Goal of the research

As already stated in the introduction, our goal is to define an abstract interpretation framework for timed concurrent constraint languages, with the aim of verifying complex concurrent and reactive systems.

The definition of an appropriate concrete semantics, capable of modeling the properties of interest, is a key point in abstract interpretation (Cousot and Cousot 1977). In particular the concrete semantics has to be:

- fully-abstract w.r.t. the *tccp* small-step behavior,
- compositional and bottom-up,
- goal-independent, i.e., the fixpoint computation does not depend on the initial agent but only on the process declarations, and
- as condensed as possible, i.e., it should not contain redundant elements.

These requirements, especially the condensedness, are particularly relevant to speed up convergence of the approximated semantics fixpoint computation and to improve the precision of the approximation. As explain in Section 2, non-determinism, local variables and timing constructs which handle negative information significantly complicates the definition of a compositional and condensed semantics. The solutions to all the mentioned difficulties (for *ccp* and its time extensions) have been traditionally based on the introduction of restrictions on the language. However, since we are interested in applying the semantics to develop (semantics-based) program manipulation tool – like debuggers, verifiers and analyzers of complex temporal, concurrent and reactive systems – for us this solution is simply not feasible. Furthermore, we are interested in modeling the behavior of reactive systems. As said before, these systems strongly depend on time and interact continuously with the environment for an infinite period. Thus, we need a semantics that is able to deal also with infinite computations, and this adds another difficulty to the definition of a fully-abstract model. All these requirements are crucial to design efficacious semantic-based debugging and verification tools based on abstract interpretation.

Another important issue is the choice of a suitable abstract domain in which the approximated semantics is evaluated. It is important to find a good trade-off between the precision of errors that can be detected and the effort in providing the specification. Obviously, if we use more abstract domains we lose precision and we can detect less errors, but the specifications are shorter and easier to be written. On the contrary, by using more precise (or more concrete) abstract domains we gain in precision but the specification become bigger and more error-prone.

4 Current status of the research

In (Comini et al. 2013a) we have developed a new (small-step) *compositional* semantics for *tccp* which is (correct and) *fully abstract* w.r.t. the small-step behavior of *full tccp*. Since *tccp* was originally defined to model reactive systems, that many times include systems that do not terminate *with a purpose*, we have developed our semantics to deal also with *non-terminating computations.*

Our idea is to associate to a *tccp* program a set of sequences (called conditional state sequences) that models the evolution of the state in a compact way. We enrich classical behavioral traces (i.e., simple sequences of constraints modeling the evolution of the constraint store) 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 constructs of a program.

In this way, we obtain a condensed semantics which deals with non-monotonicity, since into denotations we have the *minimal* information that has to be used to exploit computations arising from absence of information. Due to its compactness, our semantics is shown to be suitable for verification and debugging purposes based on abstract interpretation.

In (Comini et al. 2013a) we also define a big-step semantics which tackles also outputs of infinite computations. This semantics abstracts away from the information about the evolution of the state and keeps only the the first and the final computed global constraint store. We prove that its fragment for finite computations is (essentially) isomorphic to the traditional big-step semantics of (de Boer et al. 2000). 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 (i.e., some more information into denotations is needed).

As already said, the main aim of my research project is the definition of a fully-automatic verification method for *tccp*. Starting from our semantics, we deduce, using standard abstract interpretation techniques, an approximating semantics based on the abstraction of the underlying constraint system (Comini et al. 2011). The elements of the abstract domain are abstract compact sequences which contain approximated information and collapse in an unique state all the consecutive states that becomes equal after the abstraction. 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 (Comini et al. 1999) to automatically detect if the program meets the specification. In this way, we obtain a fully-automatic verification method for *tccp*.

Differently from the approach presented in (Falaschi et al. 2007), we do not make any restriction on the program syntax since we are able to deal also with non-monotonic operators. Furthermore, our abstraction keeps the information about infinite sequences. As claimed in Section 2, these abilities are crucial in order to model and verify interesting properties of reactive systems.

Our proposal has some important benefits: it is correct by construction and it is fully-automatic. Moreover, since the semantics is defined compositionally, all the checks are defined on each process declaration in isolation. Obviously, one cannot detect errors in declarations involving processes which have not been specified, but for the declarations that involve processes that have a specification, the check can be made, even if the whole set of declarations has not been written yet. 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. On the contrary, model checking can be applied only with a fully specified system.

As already mentioned, the main drawback of this approach (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.

In our first approach to the abstract diagnosis of *tccp* (Comini et al. 2011), the specification phase is quite error-prone, since it is necessary to list extensionally all the (abstract) conditional state sequences that represent the intended behavior of the program. To get over this problem we propose a better way to specify the desired properties of a program: by using a formula written in a suitable temporal logic as in model checking.

In (Comini et al. 2013b) we define an extension of abstract diagnosis for *tccp* (Comini et al. 2011) where the abstract domain is formed by LTL formulas. 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.

More specifically, we have defined an abstract semantics for *tccp*, by using standard abstract interpretation techniques, where the conditional state sequences of the initial semantics are approximated by means of an LTL formulas representing program properties. Intuitively, a conditional state sequence is abstracted with the most precise formula that the sequence satisfies. Then, as in the case of abstract sequences, we can apply abstract diagnosis to obtain a fully-automatic verification method that checks if a *tccp* program satisfies the 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, such check must be decidable, thus we have shown how we can instantiate the method to a sublogic (the restricted-negation fragment of CLTL presented in (Valencia 2005)) where the implication is decidable and, thus, the verification process can automatically be done. In this way, we can express most of the properties of *tccp* agents (including the non-monotonic ones) without running into the practical problems of dealing with logical negation and exploiting for our purposes the decidability results shown in (Valencia 2005).

With our proposal, we can easily specify a possible intervention coming from a surrounding environment simply by an LTL formula. With model checking, this needs to be done by simulating such environment in software with an additional set of declarations.

This approach has all the advantages of the first abstract diagnosis instance for *tccp* (Comini et al. 2011). By choosing a suitable fragment of LTL logic, where this implication is decidable, one can automatically detect the errors in the program. Furthermore, differently from the approach presented in (Valencia 2005), we do not need to restrict the language to the locally-independent fragment since our semantics is able to deal with the full language.

5 Open issues and expected achievements

Since we have achieved many of the research goals of my thesis, the most important open issue is to demonstrate the effectiveness of our approach also in practice. Thus, we will certainly implement the proposed abstract diagnosis methods, in order to be able to compare with other tools. In fact, we are convinced that our approach can be an alternative or complementary technique to model checking.

We also plan to explore other instances of the method based on logics for which decision procedures or (semi)automatic tools exists, in order to express and verify more complex properties of concurrent and reactive systems.

Our verification framework can also be immediately adapted to other concurrent (non-monotonic) languages (like *tcc* and *ntcc*) once it has been developed a suitable fully abstract semantics for them.

References

Alpuente, M., Falaschi, M., and Villanueva, A. 2003. Symbolic model checking for timed concurrent constraint programs. In *Actas de las III Jornadas de Programación y Lenguajes (PROLE'03).* Alicante (Spain), 151–165.

Alpuente, M., Gallardo, M., Pimentel, E., and Villanueva, A. 2005a. A Semantic

Framework for the Abstract Model Checking of tccp Programs. *Theoretical Computer Science 346,* 1, 58–95.

Alpuente, M., Gallardo, M., Pimentel, E., and Villanueva, A. 2005b. Abstract Model Checking of tccp programs. *Electr. Notes Theor. Comput. Sci. 112*, 19–36.

Bergstra, J. A. and Klop, J. W. 1985. Algebra of communicating processes with abstraction. *Theor. Comput. Sci. 37*, 77–121.

Berry, G. 2000. The foundations of Esterel. In *Proof, Language, and Interaction*, G. D. Plotkin, C. Stirling, and M. Tofte, Eds. The MIT Press, 425–454.

Caspi, P., Pilaud, D., Halbwachs, N., and Plaice, J. A. 1987. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, New York, NY, USA, 178–188.

Clarke, E. M., Grumberg, O., and E., L. D. 1996. Model checking. In *NATO ASI DPD.* 305–349.

Comini, M., Levi, G., Meo, M. C., and Vitiello, G. 1999. Abstract Diagnosis. *Journal of Logic Programming 39,* 1-3, 43–93.

Comini, M., Titolo, L., and Villanueva, A. 2011. Abstract Diagnosis for Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming 11,* 4-5, 487–502.

Comini, M., Titolo, L., and Villanueva, A. 2013a. A Condensed Goal-Independent Bottom-Up Fixpoint Modeling the Behavior of tccp. Tech. rep., DSIC, Universitat Politècnica de València. Available at http://riunet.upv.es/handle/10251/8351.

Comini, M., Titolo, L., and Villanueva, A. 2013b. Abstract Diagnosis for tccp using a Linear Temporal Logic.

Cousot, P. and Cousot, R. 1977. 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.* ACM Press, New York, NY, USA, 238–252.

de Boer, F. S., di Pierro, A., and Palamidessi, C. 1995. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science 151*, 37–78.

de Boer, F. S., Gabbrielli, M., and Meo, M. C. 2000. A Timed Concurrent Constraint Language. *Information and Computation 161,* 1, 45–83.

de Boer, F. S., Gabbrielli, M., and Meo, M. C. 2001. 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).* IEEE Computer Society, Washington, DC, USA, 227.

de Boer, F. S., Gabbrielli, M., and Meo, M. C. 2002. Proving correctness of Timed Concurrent Constraint Programs. *CoRR cs.LO/0208042.*

Falaschi, M., Olarte, C., and Palamidessi, C. 2009. A framework for abstract interpretation of timed concurrent constraint programs. In *PPDP*, A. Porto and F. Javier López-Fraguas, Eds. ACM, 207–218.

Falaschi, M., Olarte, C., Palamidessi, C., and Valencia, F. D. 2007. Declarative Diagnosis of Temporal Concurrent Constraint Programs. In *Logic Programming, 23rd International Conference, ICLP 2007, Proceedings*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer-Verlag, 271–285.

Falaschi, M. and Villanueva, A. 2006. Automatic verification of timed concurrent constraint programs. *Theory and Practice of Logic Programming 6,* 3, 265–300.

Gautier, T. and Le Guernic, P. 1987. SIGNAL: A declarative language for synchronous programming of real-time systems. In *FPCA*, G. Kahn, Ed. Lecture Notes in Computer Science, vol. 274. Springer, 257–277.

Hoare, C. A. R. 1978. Communicating sequential processes. *Commun. ACM 21,* 8, 666–677.

Manna, Z. and Pnueli, A. 1992. *The temporal logic of reactive and concurrent systems - specification.* Springer.

Milner, R. 1980. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science, vol. 92. Springer-Verlag, Berlin.

Milner, R., Parrow, J., and Walker, D. 1992a. A calculus of mobile processes, i. *Inf. Comput. 100,* 1, 1–40.

Milner, R., Parrow, J., and Walker, D. 1992b. A calculus of mobile processes, ii. *Inf. Comput. 100,* 1, 41–77.

Nielsen, M., Palamidessi, C., and Valencia, F. D. 2002. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing 9,* 1, 145–188.

Olarte, C., Palamidessi, C., and Valencia, F. D. 2007. Universal timed concurrent constraint programming. In *ICLP'07: Proceedings of the 23rd international conference on Logic programming.* Springer-Verlag, Berlin, Heidelberg, 464–465.

Olarte, C. and Valencia, F. D. 2008. Universal concurrent constraint programing: symbolic semantics and applications to security. In *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC08)*, R. Wainwright and H. Haddad, Eds. ACM, 145–150.

Palamidessi, C. and Valencia, F. 2001. A temporal concurrent constraint programming calculus. In *CP*, T. Walsh, Ed. Lecture Notes in Computer Science, vol. 2239. Springer, 302–316.

Saraswat, V. A. 1989. Concurrent Constraint Programming Languages. Ph.D. thesis, Carnegie-Mellon University.

Saraswat, V. A. 1993. *Concurrent Constraint Programming.* The MIT Press, Cambridge, Mass.

Saraswat, V. A., Jagadeesan, R., and Gupta, V. 1993. Programming in Timed Concurrent Constraint Languages. In *Constraint Programming: Proceedings 1993 NATO ASI, Berlin*, B. Mayoh, E. Tyugu, and J. Penjaam, Eds. Springer-Verlag, 361–410.

Saraswat, V. A., Jagadeesan, R., and Gupta, V. 1994. Foundations of Timed Concurrent Constraint Programming. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Press, 71–80.

Saraswat, V. A., Jagadeesan, R., and Gupta, V. 1996. Timed Default Concurrent Constraint Programming. *J. Symb. Comput. 22,* 5/6, 475–520.

Saraswat, V. A., Jagadeesan, R., and Gupta, V. 1999. Timed Default Concurrent Constraint Programming. *Journal on Symbolic Computation 11*, 1–42.

Saraswat, V. A. and Rinard, M. 1990. Concurrent constraint programming. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, New York, NY, USA, 232–245.

Saraswat, V. A., Rinard, M., and Panangaden, P. 1991. The Semantic Foundations of Concurrent Constraint Programming. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, New York, NY, USA, 333–352.

Valencia, F. D. 2005. Decidability of infinite-state timed CCP processes and first-order LTL. *Theoretical Computer Science 330,* 3, 577–607.