International Journal of Software Engineering and Knowledge Engineering
Vol. 0, No. 0 (2003) 000–000
© World Scientific Publishing Company

A PROCEDURE TO TRANSLATE PARADIGM SPECIFICATIONS TO PROPOSITIONAL LINEAR TEMPORAL LOGIC AND ITS APPLICATION TO VERIFICATION

JUAN CARLOS AUGUSTO
Department of Electronics and Computer Science, University of Southampton
Southampton, Hampshire SO17 1BJ, UK
jca@ecs.soton.ac.uk

and

RODOLFO SABAS GOMEZ
University of Kent, Computing Laboratory, Department of computer Science
CT2 7NZ Canterbury, Kent, UK
rsg2@kent.ac.uk

Received (received date)
Revised (revised date)
Accepted (accepted date)

Software systems have evolved from monolithic programs to systems constructed from parallel, cooperative components, as can be currently found in object-oriented applications. Although powerful, these cooperative systems are also more difficult to verify.

We show that it is possible to automatically translate a PARADIGM specification to a Propositional Linear Temporal Logic based program. This has several interesting consequences: a) on one hand we allow a more declarative view of PARADIGM specifications, b) the resulting translation is an executable specification and c) as we show in this work it can also be used to verify correctness properties by automatic means. We think this will contribute to enhance the understanding, usability and further development of PARADIGM, and related methods like SOCCA, within both the Software Engineering and the Knowledge Engineering communities.

Keywords: Software Process, PARADIGM , Temporal Logic, Verification.

1. Introduction

PARADIGM [19] is a high-level modelling language which has been proposed to design parallel and cooperative systems. It is well known as being the sub-language of SOCCA [9] used for modelling object communication, coordination and cooperation.

SOCCA and PARADIGM have received a notable attention by both Software and Knowledge Engineering communities. For example, SOCCA has been used to describe an adaptive software process [20] and an industrial maintenance case study [8], which are typical problems in the area of SE, as well as to describe Blackboard Systems [18], which is a problem in the area of Knowledge Engineering. It can also be argued that PARADIGM and SOCCA are likely to find important applications in the area of multi-agent systems. For example, [1] shows an UML statechart-based design of multi-agent systems which can be benefited by the coordination-modelling capabilities of PARADIGM . Other works (see for example, [11]) can be consulted for details about multi-agent coordination.

Propositional Linear Temporal Logic (PLTL) has been used in the specification of dynamic systems and verification of their behavior correctness ([15] and [17]). Different specification and verification systems have been proposed in the literature, notably STeP [5] and SPIN [13]. In the STeP framework SPL can be used to specify a system that is translated to a Fair Transition System. Then, behavior properties expressed by temporal logic formulas can be verified using a deductive approach. In the SPIN framework a system is specified using the Promela language to represent a system conceived through a Global State Automata. Then temporal logic formulas can again be verified but in this case using the model checking technique [4]. Other approaches to verification are based on more complex temporal assumptions like branching time, e.g., Kronos [14], here we focus on linear time leaving verification over branching time and other issues for future exploration.

We show it is possible to automatically translate a PARADIGM model into a PLTL-based program, thus obtaining an executable specification of the real system. This program will be composed by a number of logic rules implying, at any time, the current state of process executions. These rules can be entirely generated from the information provided in any PARADIGM model.

One benefit that can be expected from such a translation is that the temporal logic framework allows us to prove correctness properties by automatic means. Properties are expressed as queries to a PLTL interpreter with the logic program as a knowledge base. One such implementation of a deductive system we used for our proposal is ETP [7], which provides interpretation for a subset of PLTL covering more of the properties discussed at the end of this article. This program can also be used as a simulation tool: process executions can be traced to any situation of interest. This feature can be useful in the design stage of the software development: we can change the PARADIGM model, translate it to a logic program, and study the process behavior until functional system requirements have been met. Finally, the logic approach offers a different, declarative way to study PARADIGM models. We think this new results will contribute to enhance the understanding, usability and further development of PARADIGM, and related methods like SOCCA, within both the Software Engineering and the Knowledge Engineering communities.

This article is an extended and improved version of [2] and is organized as follows. Section 2 explains the main concepts about PARADIGM models. Section 3 explains the logic framework (PLTL) we use to specify the outcome of the translation process. The translation process itself is conceptually explained in Section 4. In Section 5 we show an algorithm which can be used to implement the translator, including a complexity analysis to show that the algorithm runs in polynomial time. Section 6 shows how the resulting translation can be used to verify correctness of behavior by quering the PLTL program with temporal properties. Conclusions are given in section 7. Due to space constraints we were forced to omit interesting sections as those exemplifying the translation procedure. The reader is kindly invited to see the whole exercise in the full version [3].

2. PARADIGM

PARADIGM models a dynamic system as a set of parallel processes. Processes are modelled as state transition diagrams (std from now on), and they can be regarded as employees or managers. Managers coordinate their employees by prescribing them a proper set subprocesses.

A subprocess is a temporal constraint placed on the employee behavior. It is modelled as an std which inherits a subset of employee's states and transitions, meaning that as long as this subprocess is prescribed the employee can only achieve part of its complete behavior. Because any employee can be controlled by several managers, its behavior at anytime results from the composite behavior assigned by each of its currently prescribed subprocesses. For example, employees can only take transitions which are included in all subprocesses currently prescribed to them. For the sake of simplicity, we have assumed that all processes of the PARADIGM model are always active.

Traps model those states in execution where employees need coordination. They are defined as subsets of subprocess states. Once an employee enters the first state of those defining a trap, the manager which prescribed the subprocess containing that trap is notified, and the employee can now only perform transitions that are inside the trap.

Manager states are assigned a set of subprocesses, one per employee. This set is currently prescribed as long as the manager remains on that state, but the same subprocess can be prescribed in several manager states. A manager cannot prescribe, at a given time, more than one subprocess per employee. Manager transitions are assigned a set of traps, meaning that transitions can only be taken if all employees are currently inside these traps. Employee executions cannot proceed outside of traps until the manager prescribes the proper set of subprocesses, thus changing their behavioral restrictions, and in the other way managers cannot proceed until the proper employees are inside their traps. An interesting example of a PARADIGM model is explained in [9].

3. The Temporal Logic

This section is devoted to introduce the temporal language to be used later for the specification of temporal properties. We just give a short introduction to the temporal logic layer of this proposal. More details about the formal theory, its use to extend Prolog with temporal operators and the algorithm used to implement an interpreter for the resulting language, ETP, can be found in [7].

Here we conceive the dynamic of the system specified with PARADIGM as a discrete sequence of steps associated to a linear conception of time ordered under the relation $\leq$. The system being specified will then evolve along a sequence of states $\sigma = s_0, s_1, \ldots$ where s_0 is the initial state. The system can or cannot have a final state s_f, allowing the consideration of reactive systems, a class of systems PARADIGM is well equipped to deal with. Each state s_i is defined by a set of atomic propositions, those who are true at that state. A set of properties θ is assumed to hold at the initial state. After n steps a computation $\sigma = s_0, \ldots, s_n$ had gone through $|\sigma| = n + 1$ states. Time here is used to refer to the different stages the system goes through. We assume a propositional language $\mathcal{L}_{\mathcal{P}}$ based on the traditional temporal operators $\Diamond A$ (A is true in some future state) and $\Box A$ (A is always true from the next state on). To simplify we consider in this article only the future fragment. Other well known operators like $\oplus$, $\mathcal{U}$ (until) and the past fragment can be added to the proposal in the future with interesting benefits during the verification stage. The set of well formed formulas of the temporal language can be defined inductively as follows:

$$\phi = p|\neg\phi|\phi_1 \wedge \phi_2|\phi_1 \vee \phi_2|\phi_1 \rightarrow \phi_2|\Diamond\phi|\Box\phi$$

where p is an atomic proposition. We define when a proposition ϕ is true in s_t where $0 \leq t \leq |\sigma|$ in a process σ, $(\sigma, t) \models \phi$, as follows:

$(\sigma, t) \models p$ *iff* $p \in s_t$ with p atomic (where $s_t \models p$ means "p is true at s_t")
$(\sigma, t) \models \neg\phi_1$ *iff* $(\sigma, t) \not\models \phi_1$
$(\sigma, t) \models \phi_1 \vee \phi_2$ *iff* $(\sigma, t) \models \phi_1$ or $(\sigma, t) \models \phi_2$
$(\sigma, t) \models \phi_1 \wedge \phi_2$ *iff* $(\sigma, t) \models \phi_1$ and $(\sigma, t) \models \phi_2$
$(\sigma, t) \models \phi_1 \rightarrow \phi_2$ *iff* $(\sigma, t) \not\models \phi_1$ or $(\sigma, t) \models \phi_2$
$(\sigma, t) \models \Diamond\phi$ *iff* there exists $s > t : (\sigma, s) \models \phi$
$(\sigma, t) \models \Box\phi$ *iff* for all $s > t : (\sigma, s) \models \phi$

This language will give us a set of well formed formulas that is expressive enough to encode the PARADIGM specification in a declarative way. It also allows us to represent well known schema formulas [16] that can be used to query the resulting temporal logic program in order to verify correctness of behavior. Some examples of these formulas are: $\Box\phi$ (*safety*) and others from the "liveness family" like $\Diamond\phi$ (*guarantee*), $\Box(\phi_1 \rightarrow \Diamond\phi_2)$ (*response/recurrence*), $\Diamond\Box\phi$ (*persistence*) and $\Box\Diamond\phi_1 \rightarrow \Box\Diamond\phi_2$ (*progress*). The framework assumes sets of propositions whose cardinality depends on the sets of manager and employee processes, they should not be prohibitively large as modularity will demand to keep the number of processes reasonably small.

Finally, we give our temporal logic a *persistence semantics*. This means a proposition P is considered true from the time it is asserted until the time it is denied, i.e., until the time proposition $\neg P$ is explicitly asserted. This helps us to express time periods: if a given information is modelled by proposition P, and it is considered valid from time t to time $t+n$, $n \in \mathbb{N}$, then this period can be expressed by asserting P at time t and $\neg P$ at time $t+n+1$. In our system, P remains true during $t, \ldots, t+n$.

4. The translation process, conceptually

The goal of the translation process is to produce a PLTL program, $\mathcal{P}$, which simulates the behavior of the processes included in the PARADIGM model.

The evolution of process executions is mapped to the state-sequence semantics of the temporal logic. We call these states *global states* in contrast to *state changes* in stds appearing in the PARADIGM model. Every global state will be a set of propositions of three possible different schemas:

a) proposition *st*, where *st* denotes a state of a given process p, will be true anytime p remains on *st*,

b) proposition *sp*, where *sp* denotes a subprocess of a given employee e, will be true anytime *sp* remains prescribed to e, and

c) proposition *tp*, where *tp* denotes a trap of a given employee e, will be true anytime e remains inside *tp*.

We assume that all propositions denoting states, subprocesses and traps are unique. For example, propositions `cpNotChecking`, `cPs3` and `tcP3` denote, respectively, that process `checkPIN` is currently on state `NotChecking`, that subprocess `checkPIN_s3` is currently prescribed and that `checkPIN` is currently inside the trap `T-cP3`.

The logic program $\mathcal{P}$ includes a set of rules which model different aspects of the PARADIGM dynamics, either by asserting or denying the truth of propositions *st*, *sp* and *tp*. These rules will be conceptually introduced in sections 4.1 to 4.5. Rules which model transitions in employee and manager process are presented first because they are the core of the system execution simulation. State changes in PARADIGM processes can be seen as a transformation of the global state at time t, G_t, into a global state at time $t+n, n \in \mathbb{N}$, G_{t+n}.

Process transitions modify the global state in different ways, and in turn global states impose different constraints on them depending on whether they are employee or manager transitions. Therefore, transitions are modelled by rules $\Box(Pre \rightarrow \Diamond Pos)$, where Pre is a set of preconditions which must hold on G_t to allow the state change, and Pos is a set of post-conditions holding on the new global state G_{t+n}, after the change. Operator $\Diamond$ expresses the fact that a state change *will* eventually occur, but does not say *when* this will happen. So rules will only reflect the order in which states can be visited. This is due to PARADIGM's lack of information concerning the time that processes spend on particular states and the time that transitions require to be performed.

Rules composing $\mathcal{P}$ will be better explained through an example we have adapted from [9]. In [9] the ATM system is modelled in SOCCA. Strictly speaking, only

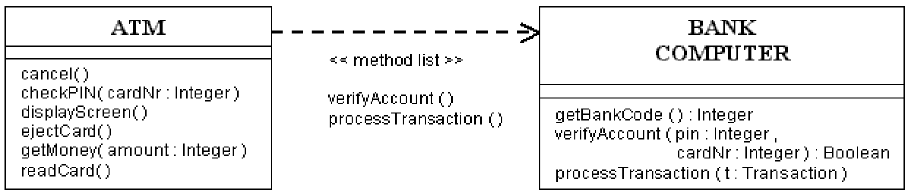

ATM
cancel()
checkPIN(cardNr : Integer)
displayScreen()
ejectCard()
getMoney(amount : Integer)
readCard()
<< method list >>
verifyAccount ()
processTransaction ()
BANK COMPUTER
getBankCode () : Integer
verifyAccount (pin : Integer, cardNr : Integer) : Boolean
processTransaction (t : Transaction)

Figure 1: Data perspective

Figs. 3 to 20 describe the PARADIGM model because it is just one of the four perspectives which are used in SOCCA to model a system. Nevertheless, we have decided to show other perspectives to make the example more readable.

The data perspective describes the static nature of a system as a collection of related classes (SOCCA is object-oriented). Fig. 1 shows two classes, `ATM` and `BankComputer` with a number of methods defining their interfaces. Also, a "use" relationship is shown describing which methods of `BankComputer` are called by `ATM` in order to perform its services. In particular, `verifyAccount()` and `processTransaction()` will be respectively called by `checkPIN()` and `getMoney()` as part of their functionality.

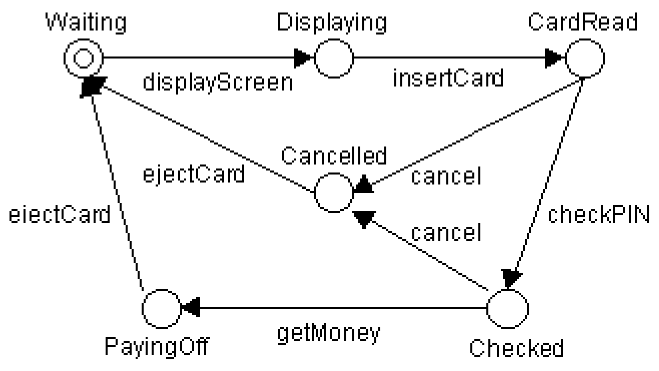
The behavior perspective describes, by means of state transition diagrams, the visible (external) behavior of the objects of a class in terms of the allowed sequence of method calling. Fig. 2 shows the behavior perspective for `ATM` and `Bankcomputer`. There we can see, for example, that `getMoney()` is never called before `checkPIN()`.

The communication perspective is specified in PARADIGM. All methods are assigned an employee process, and all classes are assigned a manager process. Each manager related to a given class `C` controls all employees related to methods of `C` plus all employees related to methods of other classes which call methods of `C`. For instance, process `checkPIN` (Fig. 3) is responsible for checking the user's magnetic card with the personal identification number, but to do this it needs to call process `verifyAccount` (Fig. 9). Both processes are employees of manager `BankComputer` (Fig. 13), which coordinates the calling-called relationship by prescribing each employee a different set of subprocesses as needed. Figs. 5 and 10 show the subprocesses that can be prescribed by BankComputer to `checkPIN` and `verifyAccount`, respectively. `checkPIN` is also employee of manager `ATM` (Fig. 20), which coordinates the operation of the ATM device. Fig. 4 shows which subprocesses `ATM` may prescribe to `checkPIN`. Traps are shown as shaded boxes.

Because this section is devoted to explain the concepts behind the translation procedure, it will be enough to comment only part of the PARADIGM model, thereby postponing the translation of the complete example to section 6.

Notational conventions are as follows. To keep graphics small we were forced to use acronyms inside figures. A guide to relate our abbreviations in figures with their references in expanded form, when used inside formulas or paragraphs: "`cP`" means "`checkPin`", "gM" means "`getMoney`", "pT" means "`processTransaction`", "vA" means "`verifyAccount`", "eC" means "`ejectCard`", "rC" means "`readCard`", "`cA`" means "`cancel`", "`T-`..." means "`Trap-`...". An example of a trap reference could be "`T-vA3`" to refer the 3rd trap in a subprocess of "`verifyAccount`". Another minor difference in notation is that when we consider the translation we are forced to write labels for the different elements of a PARADIGM specification (e.g., processes, traps, etcetera.) starting in small caps, because we feed the translation into a Prolog system and that is the way to represent propositions in Prolog programs.

class ATM



class BankComputer

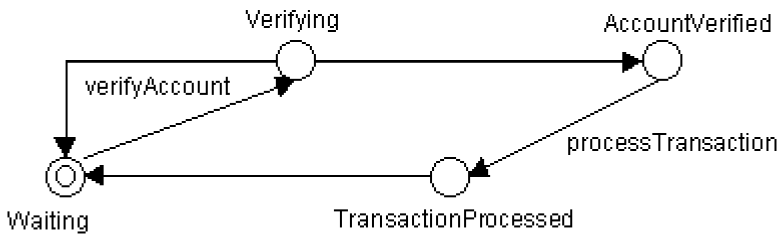


Figure 2: Behavior perspective

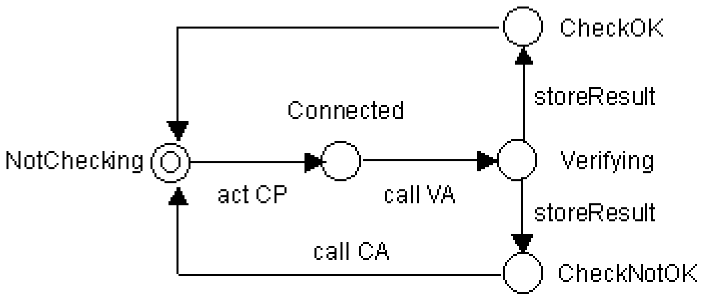


Figure 3: Employee process `checkPIN`

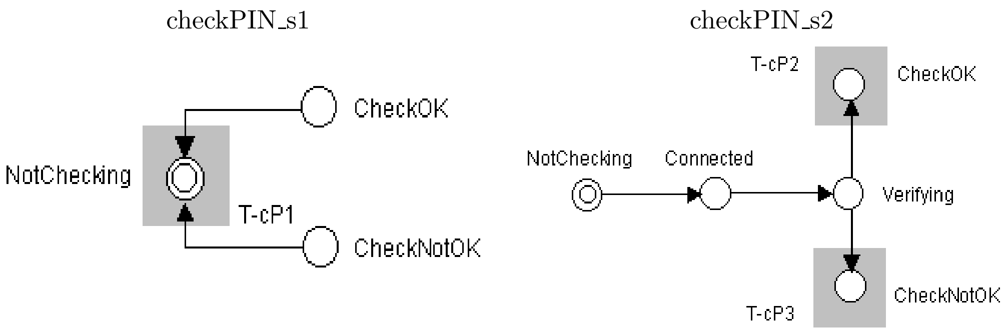


Figure 4: Subprocesses of `checkPIN` w.r.t manager `ATM`

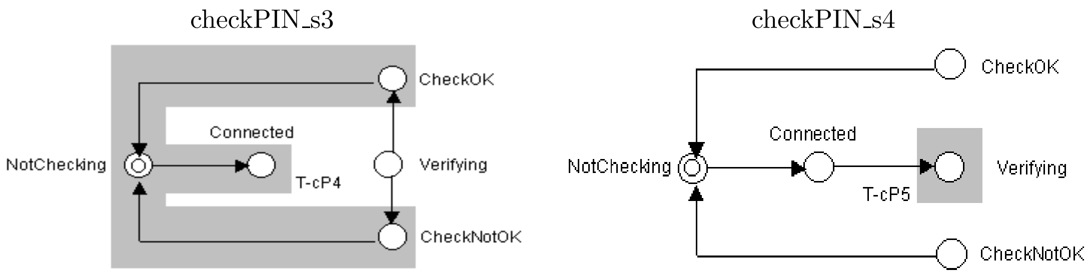


Figure 5: Subprocesses of `checkPIN` w.r.t. manager BankComputer

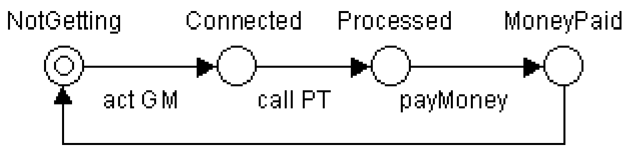


Figure 6: Employee process `getMoney`

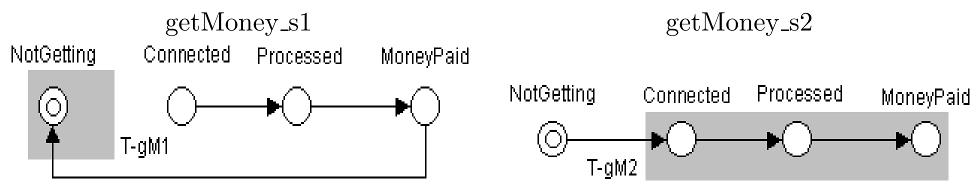


Figure 7: Subprocesses of `getMoney` w.r.t manager `ATM`

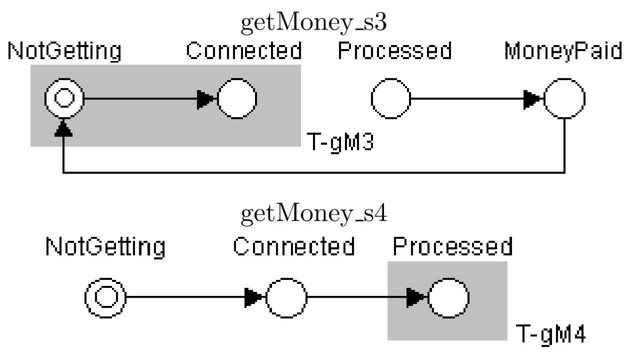


Figure 8: Subprocesses of `getMoney` w.r.t. manager BankComputer

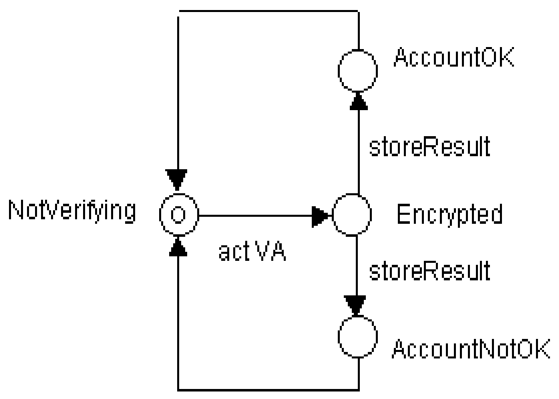


Figure 9: Employee process `verifyAccount`

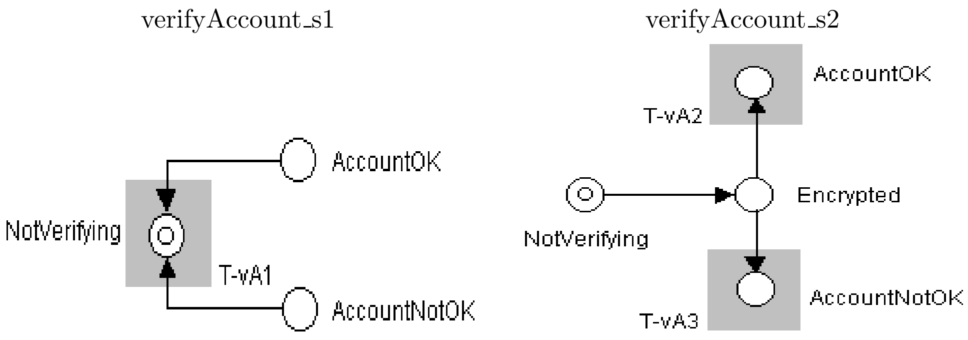


Figure 10: Subprocesses of `verifyAccount`

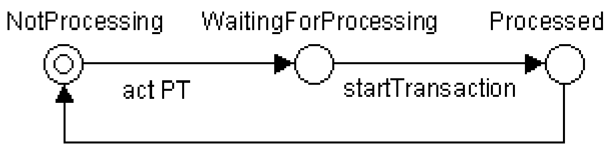


Figure 11: Employee process `processTransaction`

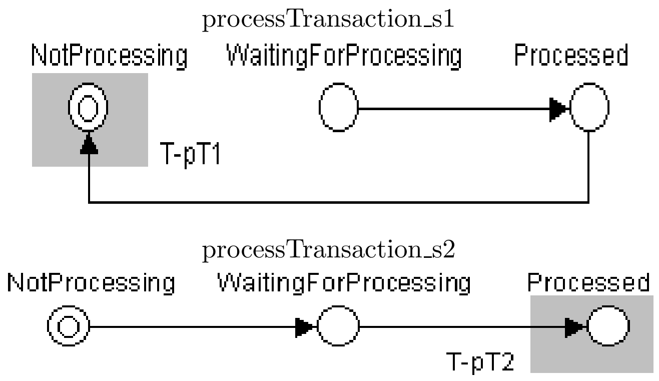


Figure 12: Subprocesses of `processTransaction`

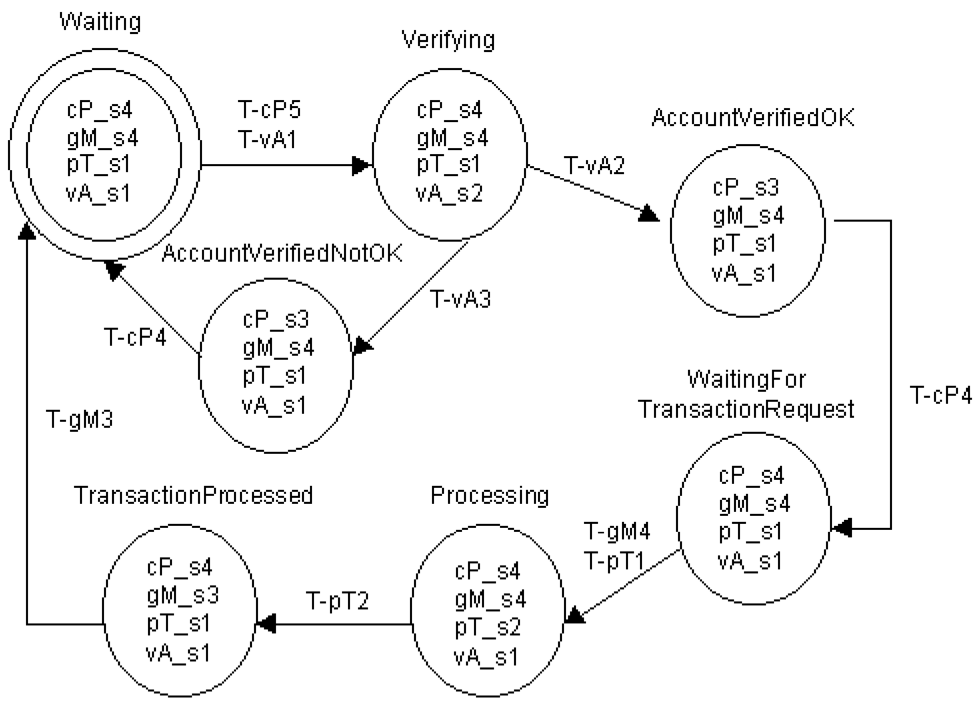


Figure 13: Manager process `BankComputer`

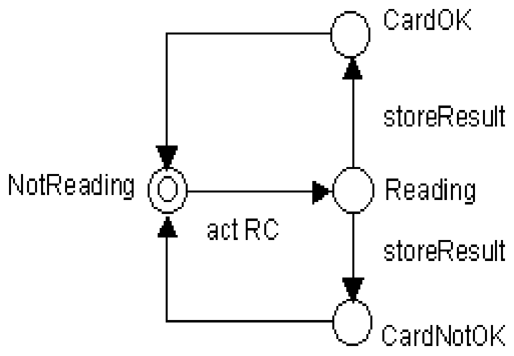


Figure 14: Employee process `readCard`

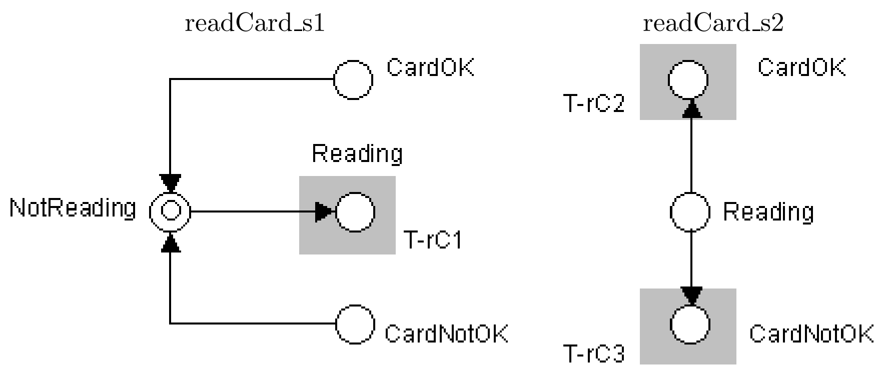


Figure 15: Subprocesses of `readCard`

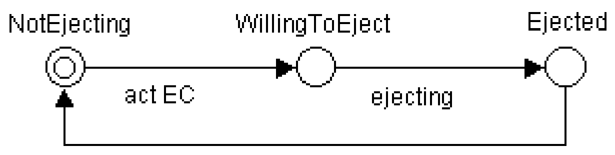


Figure 16: Employee process `ejectCard`

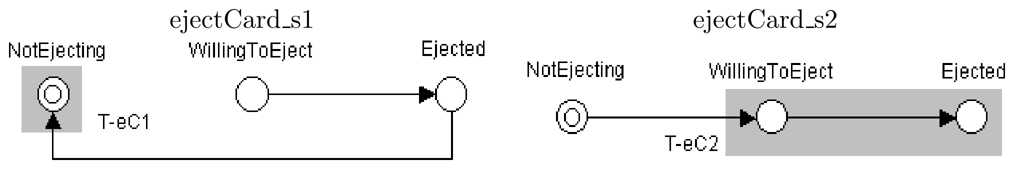


Figure 17: Subprocesses of `ejectCard`

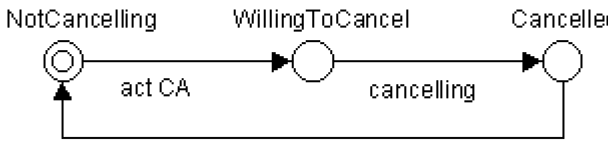


Figure 18: Employee process `cancel`

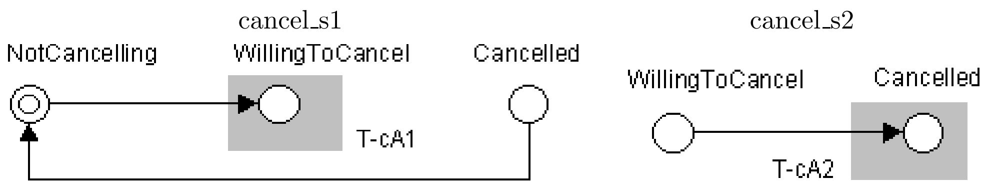


Figure 19: Subprocesses of `cancel`

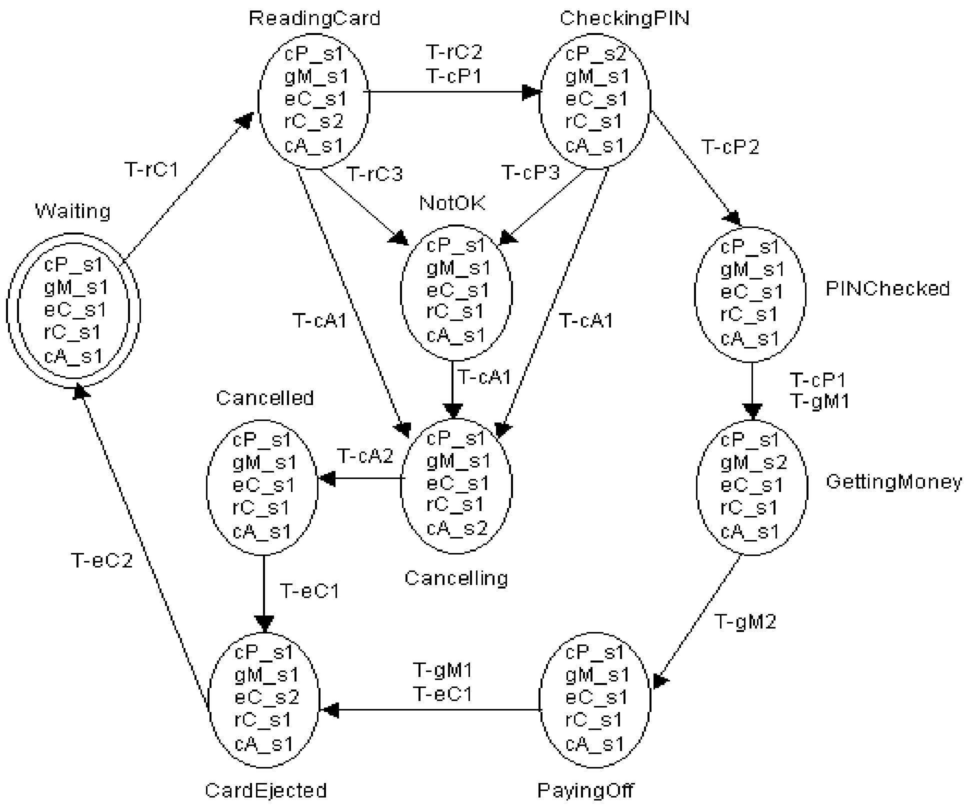

ReadingCard
cP_s1 gM_s1 eC_s1 rC_s2 cA_s1
CheckingPIN
cP_s2 gM_s1 eC_s1 rC_s1 cA_s1
T-rC2
T-cP1
T-rC1
T-rC3
T-cP3
T-cP2
Waiting
cP_s1 gM_s1 eC_s1 rC_s1 cA_s1
NotOK
cP_s1 gM_s1 eC_s1 rC_s1 cA_s1
PINChecked
cP_s1 gM_s1 eC_s1 rC_s1 cA_s1
T-cA1
T-cA1
T-cA1
T-cP1
T-gM1
Cancelled
cP_s1 gM_s1 eC_s1 rC_s1 cA_s1
T-cA2
cP_s1 gM_s1 eC_s1 rC_s1 cA_s2
Cancelling
GettingMoney
cP_s1 gM_s2 eC_s1 rC_s1 cA_s1
T-eC2
T-eC1
T-gM2
CardEjected
cP_s1 gM_s1 eC_s2 rC_s1 cA_s1
T-gM1
T-eC1
PayingOff
cP_s1 gM_s1 eC_s1 rC_s1 cA_s1

Figure 20: Manager process `ATM`

4.1. *State changes in employee processes*

These rules are related to the time each employee remains on a given state. Let ts_{ij} be a transition from state st_i to state st_j in a given employee e. This transition can be taken (at a given time t) only if:

a) e is currently on st_i and

b) ts_{ij} is included in all subprocesses currently prescribed to e

Precondition (a) can be expressed by requesting proposition st_i to be valid at t. Precondition (b) needs more explanation. Let $\mathcal{M}_e = \{m_1, \dots, m_q\}$ be the set of all managers of employee e. Of all subprocesses which can be prescribed to e by manager $m_r \in \mathcal{M}_e$, let $\mathcal{S}_r = \{sp^r_1, \dots, sp^r_n\}$ be the set of those which contains ts_{ij}. Then, ts_{ij} can be taken if for each manager m_r at least one of the subprocesses included in $\mathcal{S}_r$ is currently prescribed to e at time t. Let sets $\mathcal{S}_1, \dots, \mathcal{S}_q$ denote the subprocesses prescribed by managers $m_1, \dots, m_q$ (as defined before); containing, respectively: $\{sp^1_1, \dots, sp^1_r\}, \dots, \{sp^q_1, \dots, sp^q_s\}$. This precondition is expressed by the following conjunction:

$$((sp^1_1 \vee \dots \vee sp^1_r) \wedge \dots \wedge (sp^q_1 \vee \dots \vee sp^q_s))$$

After the state change, i.e., at time $t+n$, $n \in \mathbb{N}$, employee e will be no longer in state st_i but in st_j. This can be expressed by asserting propositions $\neg st_i$ and st_j at time $t+n$. The complete schema for rules modelling state changes in employee processes is shown next:

$$\Box((st_i \wedge (sp^1_1 \vee \dots \vee sp^1_r) \wedge \dots \wedge (sp^q_1 \vee \dots \vee sp^q_s)) \rightarrow \Diamond(\neg st_i \wedge st_j))$$

Example 1 *Here we consider a state change from* `Connected` *to* `Processed` *in employee* `getMoney`*. Fig. 7 shows that transition "*`call PT`*" is allowed in both subprocesses which can be prescribed by* `ATM`*:* `getMoney_s1` *and* `getMoney_s2`*. But Fig. 8 shows that* `getMoney_s4` *is the only subprocess, of those which can be prescribed by* `BankComputer`*, which includes "*`call PT`*". Thus, it does not matter which subprocess is* `ATM` *currently prescribing,* `getMoney_s1` *or* `getMoney_s2` *but* `BankComputer` *must be prescribing* `getMoney_s4`*. Otherwise, i.e., if* `getMoney_s3` *is currently prescribed, the state change cannot happen. To express that both, either* `getMoney_s1` *or* `getMoney_s2`*, and* `getMoney_s4` *must be prescribed we write:* `((gMs1 ∨ gMs2) ∧ gMs4)`*. The complete formula is:*

$$\Box((\texttt{gmConnected} \wedge ((\texttt{gMs1} \vee \texttt{gMs2}) \wedge \texttt{gMs4}) \rightarrow \Diamond(\neg\texttt{gmConnected} \wedge \texttt{gmProcessed}))$$

This schema can be simplified: it can be proved that the disjunction `(gMs1∨gMs2)` *is not really needed. Manager* `ATM` *is always prescribing a subprocess to* `getMoney`*, either* `getMoney_s1` *or* `getMoney_s2`*, and thus it will always be the case that one of these two subprocesses will be prescribed by the time* `getMoney` *tries to change from state* `Connected` *to* `Processed`*. Therefore the disjunction* `(gMs1 ∨ gMs2)` *is always true. In other words, a transition schema can be simplified w.r.t a given manager if the transition is included in all possible subprocesses which can be prescribed by that manager. In our example, both* `getMoney_s1` *and* `getMoney_s2` *contain transition "*`call PT`*", i.e., manager* `ATM` *can never impose a restriction on* `getMoney` *performing the change from* `Connected` *to* `Processed`*. The rule above can then be re-written as follows:*

$$\Box((\texttt{gmConnected} \wedge \texttt{gMs4}) \rightarrow \Diamond(\neg\texttt{gmConnected} \wedge \texttt{gmVerifying}))$$

This simplification strategy is included in the algorithm of section 6, step [5].

4.2. *State changes in manager processes*

The rules described in this section not only model the state changes in manager processes but also the time when subprocesses are prescribed and the time when employees are allowed to leave the traps. Let ts_{ij} be a transition from state st_i to state st_j in a given manager m. This transition can be taken (at a given time t) only if:

a) m is currently on st_i and

b) the proper employees are currently inside the traps related to ts_{ij} (see section 2)

Precondition (a) can be expressed by requesting proposition st_i to be valid at t. Let $\mathcal{T}_{entered} = \{tp_1, \ldots, tp_n\}$ be the set of traps related to transition ts_{ij}. Precondition (b) can be expressed by requesting proposition tp_i to be valid at t, for all $tp_i \in \mathcal{T}_{entered}$. After the state change, i.e., at time $t + n$,

a) m will be no longer on st_i but on st_j and

b) some of the subprocesses prescribed in st_i may not be so in st_j, and thus all traps included in them will be left.

Postcondition (a) can be expressed by asserting propositions $\neg st_i$ and st_j. Let $\mathcal{S}_{left} = \{sp_1, \ldots, sp_m\}$ be the set of subprocesses prescribed in st_i but not in st_j, and $\mathcal{T}_{left} = \{tp_q, \ldots, tp_u\}$ the set of traps included in subprocesses of $\mathcal{S}_{left}$. Postcondition (b) can be expressed by asserting $\neg sp_i$, for all $sp_i \in \mathcal{S}_{left}$, and $\neg tp_i$, for all $tp_i \in \mathcal{T}_{left}$. Finally, those subprocesses of st_i which remain prescribed in st_j and those which are prescribed only in st_j can be inferred from the assertion of proposition st_j and the rule schemas describing subprocess prescriptions in st_j (see section 4.3). Rules modelling state changes in manager processes have the following schema:

$$\Box((st_i \wedge (tp_1 \wedge \ldots \wedge tp_n)) \rightarrow \Diamond(\neg st_i \wedge st_j \wedge (\neg sp_1 \wedge \ldots \wedge \neg sp_m) \wedge (\neg tp_q \wedge \ldots \wedge \neg tp_u)))$$

Example 2 *Here we consider a state change from state* `Waiting` *to state* `Verifying` *in manager* `BankComputer` *(Fig. 13). This change cannot be performed until both traps* `T-cP5` *and* `T-vA1` *have been entered. As in state* `Waiting` *the manager is prescribing subprocesses* `checkPIN_s4` *and* `verifyAccount_s1`*. Therefore, for the manager to change to state* `Verifying` *the following conditions must hold:*

1. `checkPIN` *must be in state* `Connected`*, i.e it should have called* `verifyAccount` *(see* `checkPIN_s4` *in Fig. 5), and*

2. `verifyAccount` *must be in state* `NotVerifying`*, i.e., it must be ready to accept a new call (see* `verifyAccount_s1` *in Fig. 10).*

Once the manager is in state `Verifying`*, employee* `verifyAccount` *must be allowed to proceed with its execution, i.e., it must be allowed to leave trap* `T-vA1`*. Then, the manager prescribes* `verifyAccount_s2` *instead of* `verifyAccount_s1` *and trap* `T-vA1` *is left because it is included in* `verifyAccount_s1`*. Proposition* `(¬ vAs1 ∧ ¬ TvA1)` *expresses that* `verifyAccount_s1` *is no longer prescribed and trap* `T-vA1` *is left. Fig. 13 also shows that subprocess* `checkPIN_s4` *remains prescribed in state* `Verifying`*, and thus that* `checkPIN` *cannot leave trap* `T-cP5`*. This means* `checkPIN` *cannot proceed until* `verifyAccount` *finishes. Proposition* `bcVerifying` *and the rule shown in example 3 express that a new subprocess,* `verifyAccount_s2`*, is now prescribed and that* `checkPIN_s4` *remains prescribed. The dynamic of subprocesses is further explained in the following section. The* *complete formula to represent a change from state* `Waiting` *to state* `Verifying` *in manager* `BankComputer` *is:*

$$\square((\texttt{bcWaiting} \wedge \texttt{tcP5} \wedge \texttt{tvA1}) \rightarrow \diamond(\neg\texttt{bcWaiting} \wedge \texttt{bcVerifying} \wedge \neg\texttt{vAs1} \wedge \neg\texttt{tvA1}))$$

4.3. *Subprocess prescription*

These rules relate to the time subprocesses remain prescribed. To be more specific, for every state of a manager process there will be a rule expressing the set of subprocesses which are prescribed while the manager remains on that state. Let st_i be a manager state and $\mathcal{S}_i = \{sp_1, \ldots, sp_n\}$ be the set of subprocesses prescribed in this state. The translation process will generate the following rule:

$$\square(st_i \rightarrow (sp_1 \wedge \ldots \wedge sp_n))$$

Example 3 *The formula below describes the set of subprocesses which* `BankComputer` *prescribes in states* `Waiting` *and* `Verifying` *(Fig. 13). Those whose labels have the prefices* `cP` *and* `vA` *denote subprocesses of* `checkPIN` *(Figs. 4 and 5) and* `verifyAccount` *(Fig. 10), respectively. For example* `cPs4` *denotes subprocess* `checkPIN_s4`*. We can also see that other subprocesses are prescribed, those whose labels have the prefices* `gM` *and* `pT` *are subprocesses of* `getMoney` *(Figs. 7 and 8) and* `processTransaction` *(Fig. 12), respectively.*

$$\square(\texttt{bcWaiting} \rightarrow (\texttt{cPs4} \wedge \texttt{gMs4} \wedge \texttt{pTs1} \wedge \texttt{vAs1}))$$

$$\square(\texttt{bcVerifying} \rightarrow (\texttt{cPs4} \wedge \texttt{gMs4} \wedge \texttt{pTs1} \wedge \texttt{vAs2}))$$

4.4. *Inside a trap*

These rules model the conditions for a employee to enter and remain on a given trap. Specifically, for every trap tp in subprocess sp, where sp is a subprocess of employee e, there will be a rule expressing that e is currently inside tp. Note that this information is needed by the rules which express state changes in manager processes (see section 4.3 above).

Let $\mathcal{S}_{tp} = \{st_1, \ldots, st_n\}$ be the set of states which defines trap tp. Employee e will remain inside tp as long as sp remains prescribed to e and e remains on any state $st_i \in \mathcal{S}_{tp}$. Thus, the translation will generate rules with the following schema:

$$\square((sp \wedge (st_1 \vee \ldots \vee st_n)) \rightarrow tp)$$

Example 4 *The formula below expresses that employee* `checkPIN` *remains inside trap* `T-cP2` *as long as it is prescribed subprocess* `checkPIN_s2` *and remains on states* `Connected`*,* `Verifying` *or* `Checked`.

$$\square((\texttt{cPs2} \wedge (\texttt{cpConnected} \vee \texttt{cpVerifying} \vee \texttt{cpChecked})) \rightarrow \texttt{tcP2})$$

4.5. *Initial conditions*

All processes are supposed to start their executions coordinately. Of course, subprocesses which are prescribed to every employee at this time are those related to the set of initial states of manager processes. Let `init` be a proposition which only holds at the initial time, and $st_1, \ldots, st_n$ the set of initial states of all processes. The translation will generate rules with the following schema:

`init`
`init` $\rightarrow$ $(st_1 \ \wedge \ldots \wedge \ st_n)$

Example 5 *Below we show the initial conditions for the processes of our example.* `NotChecking`*,* `NotVerifying` *and* `Waiting` *are the initial states of employees* `checkPIN`*,* `verifyAccount` *and manager* `BankComputer`*, respectively. State* `Waiting` *implies that the first subprocesses to be prescribed by* `BankComputer` *to* `checkPIN` *and* `verifyAccount` *are, respectively,* `checkPIN_s4` *and* `verifyAccount_s1` *(see Fig. 13 and Example 3).*

```
init
init → ( cpNotChecking ∧ vaNotVerifying ∧ bcWaiting )
```

These rules provide a coordinated start of process executions and thus enables a consistent simulation of the PARADIGM model.

Although for simplicity we assumed that the initial conditions, i.e., the initial global state G_0, are generated by the translation process, it is still possible for the user to supply this information. S/he can specify different global states G'_0, $G''_0, \ldots$, to obtain different simulations. For example, processes may be assumed to start in states other than those explicitly marked as initial in the PARADIGM model.

Also the reader may notice we just made explicit the positive information, i.e., true facts. All facts which are not explicitly mentioned in the rule above are assumed false. This is in agreement with the negation as failure semantics used in our Prolog-based translation. Naturally, a different framework could demand that the user makes negative information explicit as well.

5. The translation process as an algorithm

The translation process will be described as a set of steps that takes a PARADIGM specification as input and generates a PLTL program as output. We take the PARADIGM specification "as it is", i.e., the quality of its content is the user's responsibility. At least we expect a correct specification from a purely syntactic perspective, i.e., we assume it contains all the information needed for the algorithm to produce the PLTL program.

As a matter of fact, not all elements of the PARADIGM specification are needed to obtain an executable translation. For example, it can be noticed in section 4 that transition labels are not used to generate any rule.

Those elements which are really used comprise processes, subprocesses, states, traps and some relationships between them. They will be described as a collection of sets (section 5.1), which is a suitable form future implementations can be obtained from. Indeed, the algorithm itself will be described as an "imperative-like" pseudo-code with set-manipulation primitives (section 5.2).

5.1. *Input sets*

Next we present the sets that must be provided for the translation could be performed. They encode some elements of the PARADIGM models, but we do not assume any particular tool to construct these sets. It should be easy for any tool allowing to write a PARADIGM specification to extract the basic information in a plain ASCII file with the information we need as input to our procedure.

We have chosen a set of labels to denote process, states, subprocess and traps which may differ from those appearing in the figures. However, these labels are quite obvious and easy to recognize. In some cases, they were needed to ensure uniqueness. For example, both employees `checkPIN` and `getMoney` have a state named `Connected` (Figs. 3 and 6), so we have renamed each state with a prefix denoting the process it belongs to: `cpConnected` and `gmConnected` respectively.

We can also see that subprocess labels can be quite long. Thus we have renamed them with the prefix of the process their belong to and the number of subprocess. For example, `cPs4` denotes subprocess `checkPIN_s4`. When in doubt, the reader can review the notational explanation given in page 5.

Due to space restrictions we are forced to provide a general description and just a few samples of the elements for each set. The complete input sets are given in [3].

1. A finite set EMP denoting all employee processes. In the example (Figs. 3, 6, 9 and 11) we have:

   $EMP = \{$ `checkPIN, getMoney, verifyAccount,`
   `processTransaction, readCard, ejectCard, cancel` $\}$

2. A finite set MAN denoting all manager processes. In the example (Figs. 13 and 20) we have:

   $$MAN = \{\texttt{atm}, \texttt{bankComputer}\}$$

3. A finite set $PRO_{transitions}$ denoting the set of transitions of every process:

   $$PRO_{transitions} = \bigcup_{i=1}^{n} \{(p_i, \bigcup_{j=1}^{m} \{(st_j, st_k)\})\}$$

   for some $1 \leq k \leq m$, such that p_i denotes a process and (st_j, st_k) denotes a transition from state st_j to state st_k in process p_i. In the example (Figs. 3, 6, 9, 11, 13, 14, 16, 18 and 20) we have:

```
PROtransitions = {
(checkPIN, { (cpNotChecking, cpConnected),
             ...
             (cpCheckNotOK, cpNotChecking) }),
             ...
(bankComputer, { (bcWaiting, bcVerifying),
                 ...
                 (bcAccountVerifiedNotOK, bcWaiting) })}
```

4. A finite set $MAN_{subprocesses}$ denoting the set of subprocess prescribed in every manager state:

   $$MAN_{subprocesses} = \bigcup_{i=1}^{n} \{(st_i, \bigcup_{j=1}^{m} \{sp_j\})\}$$

   where st_i denotes a manager state and sp_j denotes a subprocess prescribed in st_i. In the example (Figs. 13 and 20) we have:

```
MANsubprocesses = {
(atmWaiting, {cPs1, gMs1, eCs1, rCs1, cAs1}),
...
(bcAccountVerifiedNotOK, {cPs3, gMs4, pTs1, vAs1})}
```

5. A finite set TRP_{states} denoting the set of states defining every trap:

   $$TRP_{states} = \bigcup_{i=1}^{n} \{(tp_i, \bigcup_{j=1}^{m} \{st_j\})\}$$

where tp_i, denotes a trap and st_j denotes a state inside the trap tp_i. In the example (Figs. 3 to 12 and 14 to 19) we have:

```
TRP_states = {
(tcP1, {cpNotChecking}),
...
(tcP4, {cpNotChecking, cpConnected, cpCheckOK, cpCheckNotOK}),
...
(tcA2, {caWillingToCancel, caCancelled})}
```

6. A finite set SPR_{traps} denoting the set of traps of every subprocess:

$$SPR_{traps} = \bigcup_{i=1}^{n} \{(sp_i, \bigcup_{j=1}^{m} \{tp_j\})\}$$

where sp_i denotes a subprocess and tp_j denotes a trap of sp_i. In the example (Figs. 4, 5, 7, 8, 10, 12, 15, 17 and 19) we have:

```
SPR_traps = {
(cPs1, {tcP1}), (cPs2, {tcP2, tcP3}), (cPs3, {tcP5}), (cPs4, {tcP5}),
...
(cAs1, {tcA1}), (cAs2, {tcA2}) }
```

7. A finite set $EMP_{subprocesses}$ denoting, for every employee, the set of subprocesses which can be prescribed by every manager:

$$EMP_{subprocesses} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} \{(e_i, m_j, \bigcup_{k=1}^{q} \{sp_k\})\}$$

where e_i denotes an employee, m_j denotes a manager for e_i and sp_k denotes a subprocess of e that can be prescribed by m_j. In the example (Figs. 4, 5, 7, 8, 12, 15, 17 and 19) we have:

```
EMP_subprocesses = {
(checkPIN, atm, {cPs1, cPs2}),
(checkPIN, bankComputer,{cPs3, cPs4}),
...
(cancel, atm, {cAs1, cAs2})}
```

8. A finite set INI_{states} denoting the initial state of every process:

$$INI_{states} = \bigcup_{i=1}^{n} \{st_i\}$$

where st_i denotes the initial state of a process. In the example (Figs. 3, 6, 9, 11, 13, 14, 16 and 18) we have:

```
INI_states = { cpNotChecking, gmNotGetting, vaNotVerifying,
               ptNotProcessing, rcNotReading, ecNotEjecting,
               caNotCancelling, atmWaiting, bcWaiting }
```

9. A finite set $TRS_{subprocesses}$ denoting, for every employee transition, the set of subprocesses it is included in:

$$TRS_{subprocesses} = \bigcup_{i=1}^{n} \{((st_i, st_j), \bigcup_{k=1}^{m} \{sp_k\})\}$$

for some $1 \leq j \leq n$, where (st_i, st_j) denotes a transition of a given employee e from state st_i to state st_j and sp_k denotes a subprocess of e containing such a transition. In the example (Figs. 4, 5, 7, 8, 10, 12, 15, 17 and 19) we have:

```
TRS_subprocesses = {
((cpNotChecking, cpConnected),{cPs2, cPs3, cPs4}),
((cpConnected, cpVerifying), {cPs2, cPs4}),
...
((caCancelled, caNotCancelling), {cAs1})}
```

10. A finite set MAN_{traps} denoting the set of traps that must be entered for every state change in a manager process could be performed:

$$MAN_{traps} = \bigcup_{i=1}^{n} \{((st_i, st_j), \bigcup_{k=1}^{m} \{tp_k\})\}$$

for some $1 \leq j \leq n$, where (st_i, st_j) denotes a transition of a given manager m from state st_i to state st_j and tp_k denotes a trap that must be entered for such a transition could be performed. In the example (Figs. 13 and 20) we have:

```
MAN_traps = {
((atmWaiting, atmReadingCard), {trC1}),
((atmReadingCard, atmChekingPIN), {trC2, tcP1}),
...
((bcAccountVerifiedNotOK, bcWaiting), {tcP4})}
```

5.2. *Steps*

Now we describe the translation algorithm as a set of steps, each one taking one or more input sets (section 5.1) and generating a kind of rule for the PLTL program. We assume the existence of a procedure `generateRule()` which performs the output of a rule to the PLTL program. All variables are considered local to each step environment. Set variables are denoted with uppercase calligraphic letters, e.g., $\mathcal{A}$. Element variables are denoted with uppercase italic letters, e.g., A. Constant elements will be denoted with lowercase italic letters, e.g., a. Readers will note that some algorithm lines are distinguished with a label `[n]`, with $n > 0$. This will make sense in section 5.3 where we offer a complexity study.

1) State changes in employee processes

```
    INPUT: $EMP$, $PRO_{transitions}$, $TRS_{subprocesses}$, $EMP_{subprocesses}$
    PROCEDURE:
     % for each employee
     Tmp1 := $EMP$ ;
[1]  Repeat until Tmp1 = $\emptyset$;
     begin
        Let $e \in$ Tmp1 ;
        Tmp1 := Tmp1/$\{e\}$ ;
```

```
[2]       Let $\mathcal{T}_e$ such that $(e, \mathcal{T}_e) \in PRO_{transitions}$ ;
          % for each transition of this employee
          Tmp2 := $\mathcal{T}_e$ ;
[3]       Repeat until Tmp2 = $\emptyset$ ;
          begin
             Let $(st_i, st_j) \in$ Tmp2 ;
             Tmp2 := Tmp2/$\{(st_i, st_j)\}$ ;
             % $\mathcal{S}_{ij}$ is the set of all subprocesses containing this transition
[4]          Let $\mathcal{S}_{ij}$ such that $((st_i, st_j), \mathcal{S}_{ij}) \in TRS_{subprocesses}$ ;
             % $\mathcal{S}_e$ is the set of all subprocesses prescribed by each manager
             % to this employee
[5]          Let $\mathcal{S}_e = \{\ \mathcal{S}_m \mid \exists m \in MAN\ (\ (e, m, \mathcal{S}_m) \in EMP_{subprocesses}\ )\ \}$ ;
             % intersect each subset of $\mathcal{S}_e$ with $\mathcal{S}_{ij}$, and form the set $\mathcal{S}^m_{ij}$
             % Strict inclusion $\mathcal{I} \subset \mathcal{S}_m$ expresses the optimization
             % described in section 4.1
[6]          Let $\mathcal{S}^m_{ij} = \{\ \mathcal{I} \mid \exists \mathcal{S}_m \in \mathcal{S}_e\ (\ \mathcal{I} = \mathcal{S}_m \cap \mathcal{S}_{ij}\ \wedge\ \mathcal{I} \subset \mathcal{S}_m\ )\ \}$ ;
             Suppose $\mathcal{S}^m_{ij} = \{\ \{sp^1_1, \ldots, sp^1_r\}, \ldots, \{sp^q_1, \ldots, sp^q_s\}\ \}$ ;
[7]          GenerateRule(
             $\Box((st_i \wedge (sp^1_1 \vee \ldots \vee sp^1_r) \wedge \ldots \wedge (sp^q_1 \vee \ldots \vee sp^q_s)) \rightarrow \Diamond(\neg st_i \wedge st_j))$     )
          end % {Repeat until Tmp2 = $\emptyset$}
       end % {Repeat until Tmp1 = $\emptyset$}
```

2) State changes in manager processes.

```
  INPUT: $MAN$, $PRO_{transitions}$, $MAN_{traps}$, $MAN_{subprocesses}$
  PROCEDURE:
    % for each manager
    Tmp1 := $MAN$ ;
    Repeat until Tmp1 = $\emptyset$
    begin
      Let $m \in$ Tmp1
      Tmp1 := Tmp1/$\{m\}$ ;
      % $\mathcal{T}_m$ is the set of transitions of this manager
      Let $\mathcal{T}_m$ such that $(m, \mathcal{T}_m) \in PRO_{transitions}$ ;
      Tmp2 := $\mathcal{T}_m$ ;
      % for each transition of $\mathcal{T}_m$
      Repeat until Tmp2 = $\emptyset$ ;
      begin
         Let $(st_i, st_j) \in$ Tmp2 ;
         Tmp2 := Tmp2/$\{(st_i, st_j)\}$ ;
         % $\mathcal{T}_{ij}$ is the set traps of this transition, i.e., those traps
         % that must be entered for this transition could be performed
         Let $\mathcal{T}_{ij}$ such that $((st_i, st_j), \mathcal{T}_{ij}) \in MAN_{traps}$ ;
         % $\mathcal{I}$ is the set of subprocesses prescribed in state $st_i$
         Let $\mathcal{I}$ such that $(st_i, \mathcal{I}) \in MAN_{subprocesses}$ ;
         % $\mathcal{J}$ is the set of subprocesses prescribed in state $st_j$
         Let $\mathcal{J}$ such that $(st_j, \mathcal{J}) \in MAN_{subprocesses}$ ;
         $\mathcal{D} = \mathcal{I}/\mathcal{J}$ ;
      % $\mathcal{T}_{left}$ is the set of traps included in subprocesses of $\mathcal{D}$,
      % i.e., those traps that are left after the state change
      Let $\mathcal{T}_{left} = \{tp \mid \exists sp \in \mathcal{D} \ ( \ (sp, \mathcal{T}_{sp}) \in SPR_{traps} \ \wedge \ tp \in \mathcal{T}_{sp}) \ \}$ ;
      Suppose $\mathcal{T}_{ij} = \{tp_1, \ldots, tp_n\}$ ;
      Suppose $\mathcal{D} = \{sp_1, \ldots, sp_m\}$ ;
      Suppose $\mathcal{T}_{left} = \{tp_q, \ldots, tp_u\}$ ;
      GenerateRule(
      $\Box((st_i \wedge \ (tp_1 \wedge \ldots \wedge tp_n) \ \rightarrow$
         $\Diamond(\neg st_i \wedge st_j \wedge \ (\neg sp_1 \wedge \ldots \wedge \neg sp_m) \ \wedge \ (\neg tp_q \wedge \ldots \wedge \neg tp_u)))$ )
    end % {Repeat until Tmp2 = $\emptyset$}
  end % {Repeat until Tmp1 = $\emptyset$}
```

3) Subprocess prescriptions

```
INPUT: $MAN_{subprocesses}$
PROCEDURE:
  % for each manager state
  Tmp1 := $MAN_{subprocesses}$ ;
  Repeat until Tmp1 = $\emptyset$
  begin
    % $\mathcal{S}_{st}$ is the set of all subprocesses prescribed in this state
    Let $(st, \mathcal{S}_{st}) \in$ Tmp1 ;
    Tmp1 := Tmp1/$\{(st, \mathcal{S}_{st})\}$ ;
    Suppose $\mathcal{S}_{st} = \{sp_1, \ldots, sp_n\}$ ;
    GenerateRule( $\Box(st \rightarrow (sp_1 \wedge \ldots \wedge sp_n))$ )
  end % {Repeat until Tmp1 = $\emptyset$}
```

4) Inside a trap.

```
INPUT: $SPR_{traps}$, $TRP_{states}$
PROCEDURE:
  % for each subprocess
  Tmp1 := $SPR_{traps}$ ;
  Repeat until Tmp1 = $\emptyset$
  begin
   % $\mathcal{T}$ is the set of traps of this subprocess
   Let $(sp, \mathcal{T}) \in$ Tmp1 ;
   Tmp1 := Tmp1/$\{(sp, \mathcal{T})\}$ ;
   % for each trap in $\mathcal{T}$
   Repeat until $\mathcal{T} = \emptyset$ ;
   begin
      Let $tp \in \mathcal{T}$ ;
      $\mathcal{T}$ := $\mathcal{T}/\{tp\}$ ;
      % $\mathcal{S}_{tp}$ is the set of states defining this trap
      Let $\mathcal{S}_{tp}$ such that $(tp, \mathcal{S}_{tp}) \in TRP_{states}$ ;
      Suppose $\mathcal{S}_{tp} = \{st_1, \ldots, st_n\}$ ;
      GenerateRule( $\Box((sp \wedge (st_1 \vee \ldots \vee st_n)) \rightarrow tp)$ )
   end % {Repeat until $\mathcal{T} = \emptyset$}
  end % {Repeat until Tmp1 = $\emptyset$}
```

5) Initial conditions.

```
INPUT: $INI_{states}$
PROCEDURE:
  GenerateRule( init ) ;
  Suppose $INI_{states} = \{st_1, \ldots, st_n\}$ ;
  GenerateRule( init $\rightarrow (st_1 \wedge \ldots \wedge st_n)$ )
```

The reader can see the complete set of rules obtained from translating the example considered in this article in [3].

5.3. *Complexity*

It can be proved that our translation algorithm runs in polynomial time. We will develop our complexity analysis using the asymptotic notation often known as *"the order of"* or *"big Oh"* (see for example [6]). Thus we will find an upper bound for the worst-case execution time of the algorithm steps presented previously. Formally,

Definition 1 *Let $n \in \mathbb{N}$ be the size of the algorithm input and $t : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$ a function expressing the algorithm execution time for input n. Let $f : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$ an arbitrary function, then t is "in the order of" f iff $t(n) \in O(f(n))$, where*

$$O(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^{\geq 0} | (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N})(\forall n \geq n_0)[g(n) \leq cf(n)]\}$$

Therefore, we can state our claim in the asymptotic notation as:

Theorem 1 *Let n be the size of a given PARADIGM model $\mathcal{M}$, i.e., the input size for the translation algorithm. Let $t(n)$ be the function expressing the algorithm execution time. Let St, Sp and Tp be respectively the sets of all states, subprocesses and traps of $\mathcal{M}$. Let $IS_1, \ldots, IS_m$ be the input sets derived from $\mathcal{M}$, i.e., those sets obtained as shown in section 5.1. Then $t(n)$ is polynomial on the size of n, where $n = max(|St|, |Sp|, |Tp|, |IS_1|, \ldots, |IS_m|)$.* ▲

We defined the model size n as being the maximum cardinality among particular sets because a) the algorithm performs its computation over different input sets and b) we must operate with a unified input size to obtain a unique function expressing the order of the entire algorithm. It is also worth to mention that some execution times are considered negligible in the broader computation. These comprise assignments and the time which takes to remove an element from a set once it has already been found. In addition we assume that sets are simply implemented as lists, that all operations on sets are performed as sequential searches over their data structures and the time that takes to generate a rule is proportional to the number of propositions included in the rule schema.

The translation algorithm comprises five separate steps (see section 5.2), all assumed to be performed sequentially. Proving that each one of these steps runs in polynomial time allows us to infer the entire algorithm is polynomial. These partial proofs refer to some lines in the algorithm which has been marked with $[n]$. Function $max(a_1, \ldots, a_n)$ returns the maximum value among $a_1, \ldots, a_n$. $|S|$ denotes the cardinality of set S.

Lemma 1 *Rules expressing state changes in employee processes (see step 1 in section 5.2) can be generated in $O(n^5)$.* ▲

Justification 1 *The order of step 1 is*

$$O_{step1} = L_o.max(O_2, O_3) \tag{1}$$

where L_o is the number of iterations of the outer loop (line [1]),

$$L_o = |Tmp1| = |EMP| \leq n \tag{2}$$

and O_2 is the order of a search over $PRO_{transitions}$ (line [2]),

$$O_2 = n \tag{3}$$

and O_3 is the order of the inner loop (line [3]),

$$O_3 = L_i.max(O_4, O_5, O_6, O_7) \tag{4}$$

where L_i is the number of iterations of the inner loop (line [3]), $L_i = |Tmp2| = |\mathcal{T}_e|$, where $\mathcal{T}_e$ is the set of transitions in employee e,

$$L_i \leq n \tag{5}$$

and O_4 is the order of a search over $TRP_{subprocesses}$ (line [4]),

$$O_4 = n \tag{6}$$

and O_5 is the order of a search over $EMP_{subprocesses}$ (line [5]),

$$O_5 = n \tag{7}$$

and O_6 is the order of the time that takes to compose set $\mathcal{S}_{ij}^M$ (line [6]), which involves an intersection-inclusion proof for every element of set $\mathcal{S}_e$,

$$O_6 = |\mathcal{S}_e|.max(O_\cap, O_\subset) \tag{8}$$

where $O_\cap$ is the order of the time that takes to perform $\mathcal{S}_m \cap \mathcal{S}_{ij}$, which in turn can be bounded by $|\mathcal{S}_m|.|\mathcal{S}_{ij}|$. As $|\mathcal{S}_m|$ is at most the maximum number of subprocesses that can be prescribed by a manager to a single employee, and $|\mathcal{S}_{ij}|$ is at most the maximum of subprocesses a given transition is part of, then $|\mathcal{S}_m| \leq n$ and $|\mathcal{S}_{ij}| \leq n$, then

$$O_\cap = n^2 \tag{9}$$

and $O_\subset$ is the order of the time that takes to perform $\mathcal{I} \subset \mathcal{S}_m$, which in turn can be bounded by $|\mathcal{I}|.|\mathcal{S}_m|$. As $|\mathcal{I}|$ is at most $|\mathcal{S}_m| \leq n$, then

$$O_\subset = n^2 \tag{10}$$

and $|\mathcal{S}_e|$ is at most the maximum number of managers for a given employee,

$$|\mathcal{S}_e| \leq n \tag{11}$$

and O_7 is the order of the time that takes to generate the rule (line [7]). We can see the number of elements to be written in the PLTL program is clearly dominated by $|\mathcal{S}_{ij}^M|$, which in turn is at most $|\mathcal{S}_e| \leq n$ and then

$$O_7 = n \tag{12}$$

From eqs. 9, 10, 11 and 12 we have that $O_6 = n^3$ (eq. 8).
From eqs. 5, 6, 7 and 8 we have that $O_3 = n^4$ (eq. 4).
From eqs. 2, 3 and 4 we have that $O_{step1} = n^5$ (eq. 1).

Due to space constraints we just list the other lemmas that have to be proved in order to support our claim, omitting their proofs. The complete proof to our claim can be seen in [3].

Lemma 2 *Rules expressing subprocess prescriptions in manager states (see step 2 in section 5.2) can be generated in $O(n^2)$.* ▲

Lemma 3 *Rules expressing state changes in manager processes (see step 3 in section 5.2) can be generated in $O(n^4)$.* ▲

Lemma 4 *Rules expressing state changes in manager processes (see step 4 in section 5.2) can be generated in $O(n^3)$.* ▲

Lemma 5 *Rules expressing initial conditions (see step 5 in section 5.2) can be generated in $O(n)$.* ▲

From lemmas 1, 2, 3, 4 and 5 it can be proved that the entire translation algorithm runs in polynomial time. In fact, it is at most $O(n^5)$.

6. Model verification

It is possible to link the output of our translation, a PLTL-based program, to a procedure to verify correctness in the initial PARADIGM specification. Here we show that well-known properties from the systems verification literature [16] can be naturally associated to this translation. A later stage in our research will involve to link this notions to the already available tools SPIN and STeP.

Before offering a number of examples for such properties, our notation must be explained. Propositions `vaAccountOK`, `vaAccountNotOK` and `vaNotVerifying` are true anytime `verifyAccount` (Fig. 9) remains on states `AccountOK`, `AccountNotOK` and `NotVerifying`, respectively. Propositions `cpVeryfing`, `cpChecked` and `cpNotChecking` are true anytime `checkPIN` (Fig. 3) remains on states `Veryfing`, `Checked` and `cpNotChecking`, respectively. Propositions `TcP4`, `TvA2` and `TvA3` are true anytime `checkPIN` (Fig. 5) and `verifyAccount` (Fig. 10) remain inside traps `T-cP4`, `T-vA2` and `T-vA3`, respectively. Proposition `vAs2` is true anytime subprocess `verifyAccount_s2` (Fig 10) is prescribed.

Example 6 *(A safety property) "Any account can be either accepted or rejected, but it can never be in both states"*

$$\Box\neg(\texttt{vaAccountOK} \wedge \texttt{vaAccountNotOK})$$

Example 7 *(A guarantee property) "It is possible that the ATM reports a PIN as checked while BankComputer is still verifying it"*

$$\Diamond(\texttt{cpChecked} \wedge \texttt{vAs2})$$

Example 8 *(Some response properties) "Whenever the system reaches the state Verifying during the checkPIN stage of the procedure it eventually reaches the state where the PIN is already Checked."*

$$\Box(\texttt{cpVerifying} \rightarrow \Diamond\texttt{cpChecked})$$

"If ATM requests BankComputer to verify a PIN, it always gets an answer, either positive or negative"

$$\Box(\texttt{TcP4} \rightarrow \Diamond(\texttt{TvA2} \vee \texttt{TvA3}))$$

Example 9 *(A response/recurrence property) "The stage of verifying an account implies to check whether the account is acceptable or not. After that step the process is reinitiated."*

$$\Box(\texttt{vaNotVerifying} \rightarrow \Diamond((\texttt{vaAccountOK} \vee \texttt{vaAccountNotOK}) \wedge \Diamond\texttt{vaNotVerifying}))$$

Example 10 *(A recurrence property) "The process of checking a PIN can be cyclically invoked"*

$$\Box(\Diamond\texttt{cpNotChecking} \wedge \Diamond\neg\texttt{cpNotChecking})$$

It can be seen the verification process can be set, either at the more general level of the functionality of the system (examples 6, 9 and 10) or at a subtler level of traps and subprocesses (examples 7 and 8).

Our PLTL translation can be coupled more or less easily with a PLTL interpreter, e.g., ETP [7], to verify temporal properties. Other alternatives include the consideration of systems like STeP and SPIN. As mentioned earlier, SPIN is based on model checking. Because in this technique the space of possible states of the global automata is explored the tool is restricted to finite state systems. On the other hand highly efficient algorithms made this tool very successful for industrial applications. STeP instead is a collection of tools mainly focused on a deductive approach to verification, although also provides model checking support. Being a deductive system it can deal with infinite state specifications and hence, provides better scalability than tools centered on state-exploration like SPIN.

However some further work must yet be done in order to link our proposal with either STeP and SPIN, we think that our work on making explicit the temporal relationships implicitly encoded in each PARADIGM specification may help to accomplish future goals, as finding translations from PARADIGM to SPL. It is our conjecture that Etessami's work [10] to translate an extended version of Linear Temporal Logic (LTL) to Buchi Automata can be useful to link our specification language with other verification frameworks.

We found that PLTL is a flexible language where to easily encode a wide range of distinctive features of PARADIGM, e.g., those relating traps and suprocesses. It is our conjecture that encoding these notions in other formalisms could not be so straightforward. For example, the reader must notice that model checkers cannot deal with formulas containing operators from the past fragment of PLTL. Although the encoding of these notions in Fair Transition Systems, in the case of STeP, or global automata, in the case of SPIN, is a matter of further research, we nevertheless have learnt some important insights on the dynamic aspects of PARADIGM -based specifications. They hopefully will allow us to take some other steps in order to improve the verification techniques available to Knowledge and Software Engineers using PARADIGM.

7. Conclusions and Further Work

We have introduced a translation process that takes a PARADIGM specification as input and generates a Temporal Logic based program which expresses, from a declarative approach, the dynamic behavior of such specification. This program can be used to trace process interactions. Also, it can be seen as a database that can be queried to verify a system. For example, classical properties such as guarantee, persistence, response and others can be queried to verify the correctness of a particular PARADIGM model. A translation algorithm based on set-manipulation primitives has been presented, which can be proved to run in polynomial time.

The only related work we know about, [12], considers a transition-like operational semantics for PARADIGM. This semantics is argued to be useful to compare different PARADIGM models, e.g., to prove equivalence relations. Like ours, this semantics can be translated into an executable model and used for verification purposes. However, to our current knowledge there is no translation procedure to automatically obtain this semantics from a PARADIGM model.

A very interesting issue to be considered in further work involves rule enhancement to model processes that are not always active, as it is usually the case in real systems. Translation could be also extended to express some constraints that are not included in PARADIGM models but which usually affect the system dynamics. For example, SOCCA models, which include PARADIGM models as a perspective of the system modelled, also provides information about the order in which processes are actually called.

References

1. T. Arai and F. Stolzenburg *Multiagent Systems Specification by UML Statecharts Aiming at Intelligent Manufacturing* Universität Koblenz-Landau, December (2001).
2. Juan C. Augusto and Rodolfo S. Gómez *A Temporal Logic View of Paradigm Specifications* Proceedings of Fourteenth International Conference on Software Engineering and Knowledge Engineering (SEKE02), Ischia, Italy, July (2002) 497–503.
3. Juan C. Augusto and Rodolfo S. Gómez *A Procedure to Translate Paradigm Specifications to PLTL and its Application to Verification* Technical Report. `http://www.ecs.soton.ac.uk/~jca/Par2PLTL.pdf`, January (2002), 40 pages.
4. B. Berard and M. Bidoit and A. Finkel and F. Laroussinie and A. Petit and L. Petrucci and Ph. Schnoebelen and P. McKenzie, *Systems and Software Verification (Model Checking Techniques and Tools)*, (Springer Verlag, 1999).
5. N. Bjorner and A. Browne and M. Colon and B. Finkbeiner and Z. Manna and B. Sipma and T. Uribe, Verifying Temporal Properties of Reactive Systems: A STeP Tutorial, *Formal Methods in System Design*, **16** (1999) 227–270.
6. G. Brassard and P. Bratley, *Fundamentals of Algorithmics*, (Prentice Hall, 1996).
7. M. L. Cobo and J. C. Augusto, Logical Foundations and Implementation of an Extension of Temporal Prolog, *Journal of Computer Science & Technology* **1** (1999) 22–36.
8. T. de Buntje and G. Engels and L. Groenewegen and A. Matsinger, Industrial Maintenance Modelled in SOCCA, in *Fourth International Conference on the Software Process. Proceedings* Ed. Rijn-beek (IEEE Computer Society Press, 1996) 13–26.
9. Jürgen Ebert and Luuk Groenewegen and Roger Süttenbach, *A Formalization of SOCCA* Technical Report, Universität Koblenz-Landau, (1999) 10–99.
10. K. Etessami, Stutter-invariant languages, omega-automata, and temporal logic, in *Proc. of 11th Interntational Conference on Computer-Aided Verification* (1999) 236–248.
11. R. Frozza and L. O. Alvares, Criteria for the Analysis of Coordination in Multi-agent Applications *5th International Conference on Coordination Models and Languages, YORK, UK* Eds. Farhad Arbab and Carolyn L. Talcott, Lecture Notes in Computer Science **2315**, Springer Verlag, April (2002) 158–165.
12. L. Groenewegen and E. de Vink Operational Semantics for Coordination in PARADIGM, In *5th Int. Conference on Coordination Models and Languages* Eds. F. Arbab and C. L. Talcott, LNCS **2315**, Springer Verlag, April (2002) 191-206.
13. Gerard Holzmann, The Model Checker SPIN, *IEEE Transactions on Software Engineering* **23** (1997) 279–295.
14. Sergio Yovine, Kronos: A Verification Tool for Real-Time Systems, *Springer International Journal of Software Tools for Technology Transfer* (1997).
15. Z. Manna and A. Pnuelli, The Anchored Version of The Temporal Framework, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency* (Springer Verlag, 1989) 201–284.
16. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems (Specification)*, (Springer Verlag, 1992).
17. Amir Pnueli, Deduction is forever (invited talk) in *Formal Methods'99*, Toulouse, France, September (1999).
18. C. Spruit, *Adaptive Software Process Modelling with SOCCA and PARADIGM* University of Leiden (1995).
19. M. van Steen and L. Groenewegen and G. Oosting, Parallel Control Processes: Modular Parallelism and Communication, in *Proceedings Intelligent Autonomous Systems* Eds, Hertzberger and Groen, Amsterdam, The Netherlands (1987) 562–579.
20. A. Wulms, *Blackboard Systems Modelled in SOCCA* Technical Report, Universität at Koblenz-Landau (1997).

8. *

List of Figures

1 Data perspective 5
2 Behavior perspective 6
3 Employee process `checkPIN` 6
4 Subprocesses of `checkPIN` w.r.t manager `ATM` 7
5 Subprocesses of `checkPIN` w.r.t. manager BankComputer 7
6 Employee process `getMoney` 7
7 Subprocesses of `getMoney` w.r.t manager `ATM` 8
8 Subprocesses of `getMoney` w.r.t. manager BankComputer 8
9 Employee process `verifyAccount` 8
10 Subprocesses of `verifyAccount` 9
11 Employee process `processTransaction` 9
12 Subprocesses of `processTransaction` 9
13 Manager process `BankComputer` 10
14 Employee process `readCard` 10
15 Subprocesses of `readCard` 11
16 Employee process `ejectCard` 11
17 Subprocesses of `ejectCard` 11
18 Employee process `cancel` 12
19 Subprocesses of `cancel` 12
20 Manager process `ATM` 12