An Algorithm to Translate PARADIGM specifications to PLTL in Polynomial Time

Rodolfo Sabas Gomez *
Computing Laboratory,
Dept. of computer Science,
University of Kent,
CT2 7NZ Canterbury,
Kent, United Kingdom
+44 (0) 1227 823824
rsg2@ukc.ac.uk

Juan Carlos Augusto
Department of Electronics and
Computer Science,
University of Southampton,
SO17 1BJ Southampton,
Hampshire, United Kingdom
+44 (0) 23 8059 3440
jca@ecs.soton.ac.uk

Silvia Teresita Acuña
Universidad Nacional de
Santiago del Estero,
Avenida Belgrano (S) 1912,
(4200) Santiago del Estero,
Santiago del Estero, Argentina
+54 (0) 385 4509550
silvac@unse.edu.ar

ABSTRACT

PARADIGM has recently emerged as a new language to design cooperative object-oriented systems. To our knowledge, PARADIGM temporal aspects have not been studied before.

Here we describe a polynomial algorithm to translate PARADIGM models to Propositional Linear Temporal Logic programs. The resulting program is an executable specification of the modelled system, suitable for verifying model properties. It is also a declarative view of the model. Therefore we provide a temporal framework to understand and reason about PARADIGM models behavior, and system development in general. Finally, we believe this work provides further evidence on the benefits that PARADIGM has to offer to the Software Engineering community. We complement a previous conference paper which introduced the main concepts behind the translation process and its application in system verification.

Keywords

Distributed systems, temporal logic, Specification Languages

1. INTRODUCTION

PARADIGM [14] is a high-level, visual, object-oriented modelling language to design cooperative systems. It is the sublanguage of SOCCA [8] used for modelling object communication, coordination and cooperation. PARADIGM appears as a promising approach to the design of complex systems (see e.g. [15], [7], and [1]).

Propositional Linear Temporal Logic (PLTL) has been used both in system specification and verification [13]. A number of tools have been proposed to accomplish such tasks, notably STeP [4] and SPIN [11]. In the STeP framework the specification language SPL (that means Simple Programming Language) 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. Other approaches to verification are based on more complex temporal assumptions like branching time, e.g. Kronos [16], here we focus on linear time leaving verification over branching time and other issues for future exploration.

*The author is supported by the ORS Award Scheme, UK Universities

We show that is possible to translate a PARADIGM model into a PLTL program, thus obtaining an executable specification. The resulting PLTL program will be composed by a number of logic rules implying, at any time, the current state of process executions. Furthermore, these rules can be entirely generated from the information provided in any PARADIGM model. This paper focuses on the translation algorithm (a Prolog implementation is discussed in [2]). This work complements the results presented in [3], which was mainly concerned with translation concepts and its application in verification.

One benefit that can be expected from such a translation is that the temporal logic framework allows us to prove correctness properties about the system behavior by automatic means. Properties are intended to be expressed as queries to a PLTL interpreter (see e.g. [6]) with the logic program as a knowledge base. The interpreter can be also expected to be used as a simulation tool: process executions can be traced to any situation of interest. This feature can be useful during design stages: we can change the PARADIGM model, translate it to a logic program, and study the process behavior until requirements have been met. Finally, the logic approach offers a different, declarative way for studying PARADIGM models. It provides a possible temporal framework for system development. We therefore aim to enhance the current knowledge on PARADIGM and its benefits in system development.

Paper organization: Section 2 and 3 offer the necessary overview on PARADIGM and PLTL, respectively. Section 4 presents the concepts behind the translation algorithm, which is properly introduced in section 5. Section 7 shows some examples of verification properties, and conclusions are given in section 8.

2. PARADIGM

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

A subprocess is a temporal constraint placed on the employee behavior. It is modelled as an STD which inherits a subset of employee states and transitions, meaning that as long as this subprocess is prescribed the employee can only achieve part of its complete behavior. Because an employee can be controlled by several managers, its behavior at anytime results from the composite behavior assigned by each of its currently prescribed sub-

processes. In other words, employee transitions can only be performed if they are allowed in all subprocesses that are currently prescribed to the employee in question. For simplicity, we assume all processes in the PARADIGM model are always active.

Traps model those execution stages where employees need coordination. They are defined as being a subset 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 only perform transitions within the trap.

Manager states are assigned to a set of subprocesses, one per employee. This set is currently prescribed as long as the manager remains in that state, but it is possible for a subprocess to be prescribed in several manager states. A manager cannot prescribe, at a given time, more than one subprocess per employee.

Manager transitions are assigned to a set of traps which must be entered for the transition in question to be performed. Employee executions cannot proceed outside of traps until the manager prescribes the right set of subprocesses, thus changing their behavior restrictions, and in the other way managers cannot proceed until the right employees are inside their traps. An interesting example of a PARADIGM model is explained in [8].

3. THE TEMPORAL LOGIC

The system is thought as evolving along a (possibly finite) sequence of states $\sigma = s_0, s_1, \ldots$ where s_0 is the initial state. Notice that no final state is enforced in that sequence of states. This allows the consideration of reactive systems which is a class of systems that PARADIGM is well equipped to deal with. Each state s_i is defined as a set of atomic propositions holding at that execution stage. 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 stage sequence the system goes through, so it is linear and with future unbounded. 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). Here we only consider the future fragment, which is enough to highlight verification possibilities. Other well known operators like $\bigcirc$ (next), $\mathcal{U}$ (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 (p is an atomic proposition):

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

Formula semantics is shown below w.r.t. a pair (σ, t) where $\sigma = s_0, s_1, \ldots, s_t, \ldots$:

$(\sigma,t) \models p$	*iff* $p \in s_t$ with p atomic
$(\sigma,t) \models \neg\phi_1$	*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* exists $s > t : (\sigma,s) \models \phi$
$(\sigma,t) \models \Box\phi$	*iff* for all $s > t : (\sigma,s) \models \phi$

This language provides a set of well formed formulas which is expressive enough to encode a PARADIGM model in a declarative way. It also provides the means to express well known schema formulas [12] used in system verification, like $\Box\phi$ (*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 requires sets of propositions whose cardinality depends on the sets of manager and employee processes. Modularity principles applied over the PARADIGM model should keep these sets reasonably small. Finally, we give our temporal logic a *persistence semantics*: propositions preserve their truth values until it is explicitly changed by a rule, and, unless otherwise implied, propositions are assumed to be false by default. This will be consistent with our logic programming implementation of a PLTL interpreter.

4. THE TRANSLATION

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. Process executions are 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 kinds: 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. 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}. Where $n \in \mathbb{N}$ represents the duration of the execution of the transition being considered.

We will 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 in state `NotChecking`, that subprocess `checkPIN_s3` is currently prescribed and that `checkPIN` is currently inside trap `T-cP3`. Rules in P will simulate PARADIGM dynamics, asserting or denying the truth of propositions depending on execution stages. These rules will be conceptually introduced in sections 4.1, 4.2, 4.3, 4.4 and 4.5.

Process transitions modify the global state in different ways, and in turn global states impose different restrictions on employee and manager transitions. Therefore transitions will be modelled by rules of the form: $\Box(Pre \rightarrow \Diamond Pos)$, where Pre is a set of preconditions which must hold on G_t to perform the state change, and Pos is a set of postconditions holding on the new global state G_{t+n}, after the change. So rules will only express the order in which states can be visited. This is due to PARADIGM's lack of information concerning the time that processes spend inside states and the time that transitions require to be performed.

Rules composing $\mathcal{P}$ will be illustrated through a simpler version of the example given in [8]. Figs. 1 to 6 show part of a PARADIGM model for an Automatic Teller Machine (see [2] for a more detailed example). Process `checkPIN` (Fig. 1) is responsible for checking user's PIN on his magnetic card, but to do this it needs to call process `verifyAccount` (Fig. 4). Both processes are employees of manager `BankComputer` (Fig. 6 shows a subset of the proper STD), which coordinates the caller-called relationship by prescribing each employee a different set of subprocesses as needed. Figs. 2 and 5 show the subprocesses that can be prescribed by BankComputer to `checkPIN` and `verifyAccount`, respectively. `checkPIN` is also employee of manager `ATM`, but it is not included in our example. However it is important to show (Fig. 3) which subprocesses can be prescribed to `checkPIN` by `ATM`. Traps are shown as shaded boxes.

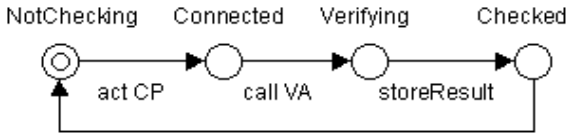
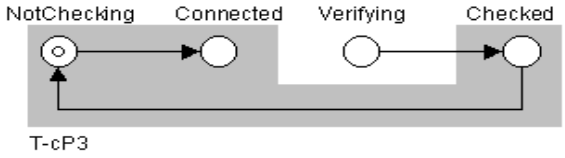


Figure 1: Employee process `checkPIN`

checkPIN_s3



checkPIN_s4

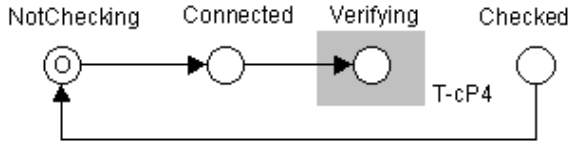
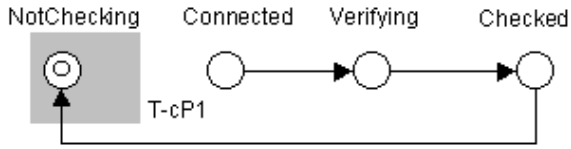


Figure 2: Sprocs. of `checkPIN` (`BankComputer`)

checkPIN_s1



checkPIN_s2

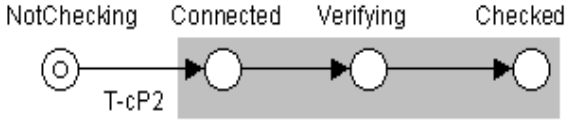


Figure 3: Sprocs. of `checkPIN` (`ATM`)

4.1. Employee transitions

These kind of rule models a given transition ts_{ij} from state st_i to st_j in employee e. This transition is allowed only if e is currently on st_i and ts_{ij} is allowed in all subprocesses that are currently prescribed to e. This set of subprocesses cannot be known in advance, but fortunately there is another way to express the same requirement. Let $M_e = \{m_1, \ldots, m_q\}$ be the set of all managers for e and $S_r = \{sp_1^r, \ldots, sp_n^r\}$ the set of all subprocesses which can be prescribed to e by any $m_r \in M_e$, such that ts_{ij} is allowed in sp_k^r, for all $1 \leq k \leq n$. Then ts_{ij} can be performed if, for each manager m_r, at least one of the subprocesses in S_r is currently prescribed. Let sets $S_1, \ldots, S_q$ denote the subprocesses prescribed by managers $m_1, \ldots, m_q$ (and constrained as mentioned before); and suppose they are, respectively: $\{sp_1^1, \ldots, sp_r^1\}, \ldots, \{sp_1^q, \ldots, sp_s^q\}$. Finally the rule expresses that after the change e will be no longer in state st_i but in st_j:

$$\Box((st_i \wedge (sp_1^1 \vee \ldots \vee sp_r^1) \wedge \ldots \wedge (sp_1^q \vee \ldots \vee sp_s^q)) \rightarrow \Diamond(\neg\, st_i \wedge st_j))$$

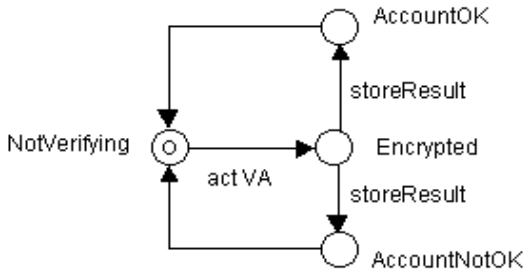
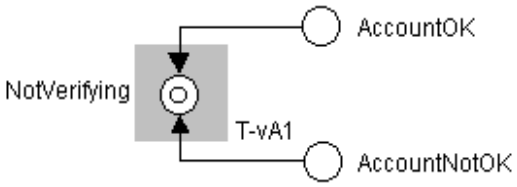


Figure 4: Employee process `verifyAccount`

verifyAccount_s1



verifyAccount_s2

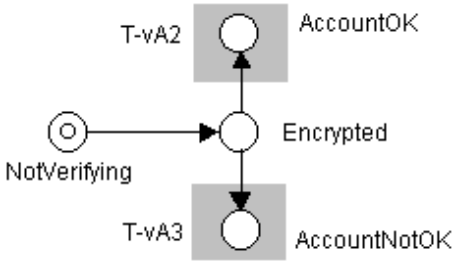


Figure 5: Subprocesses of `verifyAccount`

EXAMPLE **1** Consider transition "`call VA`" from `Connected` to `Verifying` in `checkPIN` (Figs. 3 and 2). It is allowed in `checkPIN_s1` and `checkPIN_s2` (prescribed by `ATM`), and `checkPIN_s4` (prescribed by `BankComputer`). Therefore either `checkPIN_s1` or `checkPIN_s2`, and `checkPIN_s4` must be prescribed for the transition to be performed ("(cPs1∨cPs2)∧cPs4"):

$$\Box(\texttt{(cpConnected} \wedge \texttt{((cPs1} \vee \texttt{cPs2)} \wedge \texttt{cPs4))} \rightarrow \Diamond(\neg\texttt{cpConnected} \wedge \texttt{cpVerifying}))$$ △

Note that it is also possible that other outgoing transitions from state st_i are included in the same subprocesses $S_1, ..., S_q$. Let $st_j^2, st_j^3, ..., st_j^u$ be the after states in question, and $\oplus$ the exclusive 'or' operator ($A \oplus B \equiv ((A \wedge \neg B) \vee (\neg A \wedge B))$). Then the general rule schema becomes:

$$\Box((st_i \wedge (sp_1^1 \vee \ldots \vee sp_r^1) \wedge \ldots \wedge (sp_1^q \vee \ldots \vee sp_s^q)) \rightarrow \Diamond(\neg\, st_i \wedge (st_j \oplus st_j^2 \oplus st_j^3 \oplus \ldots \oplus st_j^u)))$$

4.2. Manager transitions

These kind of rule models a given transition ts_{ij} from state st_i to st_j in manager m. This transition is allowed only if m is currently on st_i and the right employees are currently inside the traps related with ts_{ij}. This set of subprocesses cannot be known in advance, but fortunately there is another way to express the same requirement. The rule also expresses that after the transition has been performed m is no longer in st_i but in st_j, and therefore that the set of currently prescribed subprocesses

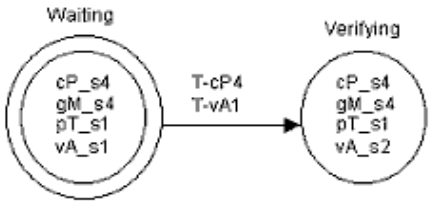


Figure 6: Manager process `BankComputer` (fragment)

has been (possibly) changed. In fact, it is possible that some of those subprocesses prescribed on st_i are no longer prescribed on st_j, or remain prescribed on st_j or even that new subprocesses are prescribed on st_j.

Let $\{tp_1, \ldots, tp_n\}$ be the set of traps related with ts_{ij}, $\{sp_1, \ldots, sp_m\}$ the set of subprocesses prescribed on st_i but not on st_j, and $\{tp_q, \ldots, tp_u\}$ the set of traps included in any $sp_i, 1 \leq i \leq m$. We will see that all new subprocesses that are prescribed in st_j but not in st_i are handled by other kind of rule (section 4.3). Finally, those propositions denoting subprocesses that remain prescribed from st_i to st_j do not require any action, i.e., they are not explicitly asserted nor denied and thus remain true (persistence semantics). The rule is shown below:

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

EXAMPLE **2** Consider the transition from `Waiting` to `Verifying` in `BankComputer` (Fig. 6). This change cannot be performed until both traps `T-cP4` and `T-vA1` have been entered ("`tcP4` $\wedge$ `tvA1`"). Once the manager is in state `Verifying`, subprocess `verifyAccount_s1` is no longer prescribed and thus trap `T-vA1` is left ("$\neg$ `vAs1` $\wedge$ $\neg$ `tvA1`"):

$$\Box((\texttt{bcWaiting} \wedge \texttt{tcP4} \wedge \texttt{tvA1}) \rightarrow \Diamond(\neg\ \texttt{bcWaiting} \wedge \texttt{bcVerifying} \wedge \neg\ \texttt{vAs1} \wedge \neg\ \texttt{tvA1}))$$

△

Previous rules model the time each employee and manager process remain in their states. However, this time depends on the dynamics of subprocesses and traps. Following rules assert that knowledge.

4.3. Subprocess prescription

This kind of rule models the time subprocesses remain prescribed to their employees. Specifically, for every state of a manager process there will be a rule expressing the set of subprocesses that are prescribed while the manager remains in that state. Let st_i be a manager state and $\{sp_1, \ldots, sp_n\}$ the subprocesses which are prescribed in this state. The rule is shown below:

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

EXAMPLE **3** Consider the subprocesses that `BankComputer` prescribes while it is in state `Waiting` (Fig. 6), like `checkPIN_s4` ("`cPs4`") and `verifyAccount_s1` ("`vAs1`"):

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

△

4.4. Traps

This kind of rule models the time employees remain inside their traps. 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 if e it is in any state $st_1, \ldots, st_n$ included in tp. Rule is shown below:

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

EXAMPLE **4** Employee `checkPIN` remains inside trap `T-cP2` as long as it is prescribed subprocess `checkPIN_s2` and remains in states `Connected`, `Verifying` or `Checked`:

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

△

4.5. Initial conditions

We provide a rule to set propositions at the first global state. Here we coordinate processes as starting execution in their initial states, but different rules may be provided to change the initial execution conditions.

Let $\{st_1, \ldots, st_n\}$ be the set of processes initial states (subprocesses initially prescribed are inferred according to the rules of section 4.3). The translation will generate the fact `init`, a proposition which will only hold at the initial time, and the following rule:

$$\texttt{init} \rightarrow (st_1 \wedge \ldots \wedge st_n)$$

EXAMPLE **5** The rule below specify the initial states of employees `checkPIN`, `verifyAccount` and manager `BankComputer`:

$$\texttt{init} \rightarrow (\texttt{cpNotChecking} \wedge \texttt{vaNotVerifying} \wedge \texttt{bcWaiting})$$

△

Note that the assertion of `bcWaiting` as initial state implies a set of initial subprocesses `cPs4, gMs4, pTs1, vAs1` by virtue of the rule shown in example 3, and in turn, because of `vAs1` and `vaNotVerifying`, this also implies that `tvA1` holds (fig. 5). Also, and according to our persistence semantics, all other propositions, i.e. the rest of states, subprocesses and traps which are not mentioned nor implied, are assumed to be false in the initial state.

5. THE ALGORITHM

In this section the translation process will be described as a set of steps that takes a PARADIGM model as input and generates a PLTL program as output. The PARADIGM model is assumed to be correct, and contains all the information needed by the algorithm: processes, subprocesses, states, traps and some of their relationships. This information is modelled as a collection of sets (section 5.1), which is a suitable form where future implementations can be obtained from. The algorithm itself will be described in natural language (section 5.2).

5.1. Algorithm inputs

The sets that are shown below encode those elements of the PARADIGM model which are needed by the translation process. We do not assume any particular tool for constructing these sets. For the sake of space economy, we have chosen a set of labels for denoting processes, states, subprocesses and traps which are shorter than those appearing in the figures. However, these labels are quite obvious and easy to recognize. For example, `cPs4` denotes subprocess `checkPIN_s4`. Input sets comprise the following (examples refer to the Automatic Teller Machine case study described in section 4):

1. The set EMP of employee processes. For example,

   EMP = {checkPIN, verifyAccount, ...}

2. The set MAN of manager processes. For example,

   MAN = {atm, bankComputer, ...}

3. The set PRO_{trs} of transitions in every process. $PRO_{trs} = \bigcup_{i=1}^{n} \{(p_i, \bigcup_{j=1}^{m} \{(st_j, st_k)\})\}$, $1 \leq k \leq m$, where p_i denotes a process and (st_j, st_k) denotes a transition from state st_j to state st_k in p_i:

   PRO_{trs} =
   {(checkPIN,
   {(cpNotChecking, cpConnected),
   (cpConnected, cpVerifying),
   (cpVerifying, cpChecked),
   (cpChecked, cpNotChecking)}),
   (verifyAccount,
   {(vaNotVerifying, vaEncrypted),
   (vaEncrypted, vaAccountOK),
   (vaEncrypted, vaAccountNotOK),
   (vaAccountOK, vaNotVerifying)})
   (vaAccountNotOK, vaNotVerifying)})
   ...}

4. The set MAN_{spr} of subprocesses prescribed in every manager state. $MAN_{spr} = \bigcup_{i=1}^{n} \{(st_i, \bigcup_{j=1}^{m} \{sp_j\})\}$ where st_i denotes a manager state and sp_j denotes a subprocess prescribed in st_i. For example,

   MAN_{spr} =
   {(bcWaiting, {cPs4, gMs4, pTs1, vAs1}),
   (bcVerifying, {cPs4, gMs4, pTs1, vAs2})
   ...}

5. The set TRP_{sta} of states defining every trap. $TRP_{sta} = \bigcup_{i=1}^{n} \{(tp_i, \bigcup_{j=1}^{m} \{st_j\})\}$ where tp_i, denotes a trap and st_j denotes a state inside trap tp_i. For example,

   TRP_{sta} =
   {(tcP1, {cpNotChecking}),
   (tcP2,
   {cpConnected,cpVerifying,cpChecked}),
   (tcP3,
   {cpNotChecking,cpConnected,cpChecked}),
   (tcP4, {cpVerifying}),
   (tvA1, {vaNotVerifying}),
   (tvA2, {vaAccountOK}),
   (tvA3, {vaAccountNotOK}),
   ...}

6. The set SPR_{trp} of traps in every subprocess. $SPR_{trp} = \bigcup_{i=1}^{n} \{(sp_i, \bigcup_{j=1}^{m} \{tp_j\})\}$ where sp_i denotes a subprocess and tp_j denotes a trap of sp_i. For example SPR_{trp} =
   {(cPs1, {tcP1}),
   (cPs2, {tcP2}),
   (cPs3, {tcP3}),
   (cPs4, {tcP4}),
   (vAs1, {tvA1}),
   (vAs2, {tvA2, tvA3})...}

7. The set EMP_{spr} of subprocesses which can be prescribed by every manager to every employee. $EMP_{spr} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} \{(e_i, m_j, \bigcup_{k=1}^{q} \{sp_k\})\}$ 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. For example

   EMP_{spr} =
   {(checkPIN,atm,{cPs1,cPs2}),
   (checkPIN,bankComputer,{cPs3,cPs4}),
   (verifyAccount,bankComputer,{vAs1,vAs2})
   ...}

8. The set INI_{sta} of initial states. $INI_{sta} = \bigcup_{i=1}^{n} \{ST_i\}$ where st_i denotes a process initial state.

   INI_{sta} =
   { cpNotChecking,vaNotVerifying,bcWaiting,
   ...}

9. The set TRS_{spr} of subprocesses every employee transition is included in.
   $TRS_{spr} = \bigcup_{i=1}^{n} \{((st_i, st_j), \bigcup_{k=1}^{m} \{sp_k\})\}$ 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. For example,

   TRS_{spr} =
   {((cpNotChecking,cpConnected),
   {cPs2,cPs3,cPs4}),
   ((cpConnected,cpVerifying),
   {cPs1,cPs2,cPs4}),
   ((cpVerifying,cpChecked),
   {cPs1,cPs2,cPs3}),
   ((cpChecked,cpNotChecking),
   {cPs1,cPs3,cPs4}),
   ((vaNotVerifying,vaEncrypted),{vAs2}),
   ((vaEncrypted,vaAccountOK),{vAs2}),
   ((vaEncrypted,vaAccountNotOK),{vAs2}),
   ((vaAccountOK,vaNotVerifying),{vAs1}),
   ((vaAccountNotOK,vaNotVerifying),{vAs1}),
   ...}

10. The set MAN_{trp} of traps related with manager transitions. $MAN_{trp} = \bigcup_{i=1}^{n} \{((st_i, st_j), \bigcup_{k=1}^{m} \{tp_k\})\}$ 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 related with that transition. For example,

    MAN_{trp} =
    {((bcWaiting,bcVerifying),{tcP4,tvA1}),
    ...}

5.2. Algorithm 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.

1) State changes in employee processes

INPUT: EMP, $PRO_{transitions}$, $TRS_{subprocesses}$, $EMP_{subprocesses}$

PROCEDURE:

```
    % for each employee
    Tmp1 := EMP
[1] Repeat until Tmp1 = ∅
    begin
      Let e ∈ Tmp1
      Tmp1 := Tmp1/{e}
[2]   Let T_e such that (e, T_e) ∈ PRO_transitions
      % for each transition of this employee
      Tmp2 := T_e
      Generated := ∅
[3]   Repeat until Tmp2 = ∅
      begin
        Let (st_i, st_j) ∈ Tmp2
        Tmp2 := Tmp2/{(st_i, st_j)}
        % S_ij is the set of all subprocesses
        % containing this transition
[4]     Let S_ij such that:
            ((st_i, st_j), S_ij) ∈ TRS_subprocesses
        % S_e is the set of all subprocesses
        % prescribed by each manager to this
        % employee
[5]     Let S_e = {S_m | ∃m ∈ MAN
                  ((e, m, S_m) ∈ EMP_subprocesses)}
        % intersect each subset of S_e with
        % S_ij, and form the set S^m_ij.
        % I ⊂ S_m expresses the optimization
        % described in sec. 4.1
[6]     Let S^m_ij = {I | ∃S_m ∈ S_e
                        (I = S_m ∩ S_ij ∧ I ⊂ S_m)}
[7]     Let J = {st'_j|((st_i, st'_j), S'_ij) ∈ TRS_subprocesses
                    ∧∀I ∈ S^m_ij(I ∩ S'_ij ≠ ∅)}
        if (st_i, J, S^m_ij) ∉ Generated
           then
           Suppose:
             S^m_ij = {{sp^1_1, ..., sp^1_r}, ..., {sp^q_1, ..., sp^q_s}}
             J = {st_j, st^2_j, st^3_j, ..., st^u_j}
[8]        GenerateRule(
             □((st_i ∧ (sp^1_1 ∨ ... ∨ sp^1_r )
                       ∧ ... ∧ (sp^q_1 ∨ ... ∨ sp^q_s))
                  →◇(¬st_i∧(st_j ⊕ st^2_j ⊕ ... ⊕ st^u_j)))
             )
           Generated:=Generated ∪{(st_i, J, S^m_ij)}
      end % {Repeat until Tmp2 = ∅}
    end % {Repeat until Tmp1 = ∅}
```

2) State changes in manager processes.

INPUT: MAN, $PRO_{transitions}$, MAN_{traps}, $MAN_{subprocesses}$

PROCEDURE:

```
% for each manager
    Tmp1 := MAN
[1] Repeat until Tmp1 = ∅
    begin
      Let m ∈ Tmp1
      Tmp1 := Tmp1/{m}

      % T_m is the set of transitions of this
      % manager
[2]   Let T_m be such that:
            (m, T_m) ∈ PRO_transitions
      Tmp2 := T_m

      % for each transition of T_m
[3]   Repeat until Tmp2 = ∅
      begin
        Let (st_i, st_j) ∈ Tmp2
        Tmp2 := Tmp2/{(st_i, st_j)}

        % T_ij is the set traps of this
        % transition, i.e. those traps that
        % must be entered for this transition
        % could be performed
[4]     Let T_ij be such that:
            ((st_i, st_j), T_ij) ∈ MAN_traps

        % I is the set of subprocesses
        % prescribed in state st_i
[5]     Let I be such that:
            (st_i, I) ∈ MAN_subprocesses

        % J is the set of subprocesses
        % prescribed in state st_j
[6]     Let J be such that:
            (st_j, J) ∈ MAN_subprocesses
[7]     D = I/J

        % T_left is the set of traps included
        % in subprocesses of D, i.e. those
        % traps that are left after the
        % state change
[8]     Let T_left = {tp | ∃sp ∈ D
                  ((sp, T_sp) ∈ SPR_traps ∧ tp ∈ T_sp)}
        Suppose T_ij = {tp_1, ..., tp_n}
        Suppose D = {sp_1, ..., sp_m}
        Suppose T_left = {tp_q, ..., tp_u}
[9]     GenerateRule(
          □((st_i∧(tp_1 ∧ ... ∧ tp_n) →
               ◇(¬st_i ∧ st_j ∧(¬sp_1 ∧ ... ∧ ¬sp_m)
                              ∧(¬tp_q ∧ ... ∧ ¬tp_u)))
          )
      end % {Repeat until Tmp2 = ∅}
    end % {Repeat until Tmp1 = ∅}
```

3) Subprocess prescriptions

INPUT: $MAN_{subprocesses}$

PROCEDURE:

```
% for each manager state
    Tmp1 := MAN_subprocesses
[1] Repeat until Tmp1 = ∅
    begin

      % S_st is the set of all subprocesses
      % prescribed in this state
      Let (st, S_st) ∈ Tmp1
      Tmp1 := Tmp1/{(st, S_st)}
      Suppose S_st = {sp_1, ..., sp_n}
[2]   GenerateRule( □(st → (sp_1 ∧ ... ∧ sp_n)) )
    end % {Repeat until Tmp1 = ∅}
```

4) Inside a trap.

INPUT: SPR_{traps}, TRP_{states}

PROCEDURE:

```
% for each subprocess
    Tmp1 := SPR_traps
[1] Repeat until Tmp1 = ∅
    begin

     % T is the set of traps of this
     % subprocess
     Let (sp, T) ∈ Tmp1
     Tmp1 := Tmp1/{(sp, T)}

     % for each trap in T
[2]  Repeat until T = ∅
     begin
        Let tp ∈ T
        T := T/{tp}

        % S_tp is the set of states defining
        % this trap
[3]     Let S_tp such that (tp, S_tp) ∈ TRP_states
        Suppose S_tp = {st_1, ..., st_n}
[4]     GenerateRule(□((sp ∧ (st_1 ∨ ... ∨ st_n)) → tp))
     end % {Repeat until T = ∅}
    end % {Repeat until Tmp1 = ∅}
```

5) Initial conditions.

INPUT: INI_{states}

PROCEDURE:

```
     GenerateRule( init )
     Suppose INI_states = {st_1, ..., st_n}
[1]  GenerateRule( init → (st_1 ∧ ... ∧ st_n) )
```

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 [5]). 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 $t : \mathbb{N} \to \mathbb{R}^{\geq 0}$ a function expressing the algorithm execution time for input n. Let $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ an arbitrary function, then t is "in the order of" f iff $t(n) \in O(f(n))$, where

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

△

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 for obtaining a unique function expressing the order of the entire algorithm.

It is also worthy to mention that some execution times are considered negligible in the broader computation. These comprise assignments and the time that 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 set operations 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 allow 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 $O(n^5)$. ▲

Sketch of proof **1** The order of step 1 is

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

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

$$L_o = |Tmp1| = |EMP| \leq n \quad (2)$$

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

$$O_2 = n \quad (3)$$

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

$$O_3 = L_i.max(O_4, O_5, O_6, O_7, O_8) \quad (4)$$

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,

$$L_i \leq n \quad (5)$$

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

$$O_4 = n \quad (6)$$

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

$$O_5 = n \quad (7)$$

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 $\mathcal{S}_e$,

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

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

$$O_\cap = n^2 \quad (9)$$

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

$$O_\subset = n^2 \quad (10)$$

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

$$|\mathcal{S}_e| \leq n \quad (11)$$

O_7 refers to the set $\mathcal{J}$ of other outgoing transitions from the same state. This construction, and related operations like the search over the `Generated` set can be proved to be subsumed by O_6.

O_8 is the order of the time that takes to generate the rule (line [8]). 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

$$O_8 = n \quad (12)$$

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). △

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

Sketch of proof **2** The order of step 2 is

$$O_{step2} = L_o.O_2 \quad (13)$$

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

$$L_o = |Tmp1| = |MAN_{subprocesses}| \leq n \quad (14)$$

and O_2 is the order of the time that takes to generate the rule (line [2]). We can see the number of elements to be written in the PLTL program is clearly dominated by —$\mathcal{S}_{st}$—, which in turn is at most the maximum number of subprocesses that a manager can prescribe on a single state, and then

$$O_2 = n \quad (15)$$

From eqs. 14 and 15 we have that $O_{step2} = n^2$ (eq. 13). △

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

Sketch of proof **3** The order of step 3 is

$$O_{step3} = L_o.max(O_2, O_3) \quad (16)$$

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

$$L_o = |Tmp1| = |MAN| \leq n \quad (17)$$

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

$$O_2 = n \quad (18)$$

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

$$O_3 = L_i.max(O_4, O_5, O_6, O_7, O_8, O_9) \quad (19)$$

where L_i is the number of iterations of the inner loop (line [3]). As $L_i = |Tmp2| = |\mathcal{T}_m)|$, where $\mathcal{T}_m$ is the set of transitions in manager m, then

$$L_i \leq n \tag{20}$$

and O_4 is the order of a search over MAN_{traps} (line [4]),

$$O_4 = n \tag{21}$$

and $O_5 = O_6$ is the order of a search over $MAN_{subprocesses}$ (lines [5] and [6]),

$$O_5 = O_6 = n \tag{22}$$

and O_7 is the order of the time that takes to compose the set $\mathcal{D}$, which in turn involves the time that takes to perform the difference $\mathcal{I}_m/\mathcal{J}(line[7])$. As this time is bounded by $|\mathcal{I}|.|\mathcal{J}|$ and $|\mathcal{I}|$ and $|\mathcal{J}|$ are at most the maximum number of subprocesses that can be prescribed by a manager to a single employee, then $|\mathcal{I}| \leq n$ and $|\mathcal{J}| \leq n$ so

$$O_7 = n^2 \tag{23}$$

and O_8 is the order of the time that takes to compose set $\mathcal{T}_{left}$ (line [8]), which involves a search over SPR_{traps} for every element of set $\mathcal{D}$. This time is bounded by $|\mathcal{D}|.|SPR_{traps}| \leq n^2$, and then

$$O_8 = n^2 \tag{24}$$

and O_9 is the order of the time that takes to generate the rule (line [9]). We can see the number of elements to be written in the PLTL program is clearly dominated by $|\mathcal{T}_{ij}| + |\mathcal{D}| + |\mathcal{T}_{left}|$. These cardinalities are at most the maximum number of employees for any manager, the maximum number of subprocesses that can be prescribed on a single manager state and the maximum number of traps in the PARADIGMmodel respectively. Therefore, the time of generation is at most $3n$ yielding

$$O_9 = n \tag{25}$$

From eqs. 20, 21, 22, 23, 24 and 25 we have that $O_3 = n^3$ (eq. 19).
From eqs. 17 and 18 we have that $O_{step3} = n^4$ (eq. 16). △

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

Sketch of proof **4** The order of step 4 is

$$O_{step4} = L_o.O_i \tag{26}$$

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

$$L_o = |Tmp1| = |SPR_{traps}| \leq n \tag{27}$$

and O_i is the order of the inner loop (line [2]),

$$O_i = L_i.max(O_3, O_4) \tag{28}$$

where L_i is the number of iterations of the inner loop, this is $L_i = |Tmp2| = |\mathcal{T}|$ where $|\mathcal{T}|$ is at most the maximum number of traps in any subprocess, and then

$$L_i \leq n \tag{29}$$

and O_3 is the order of a search over TRP_{states} (line [3]),

$$O_3 = n \tag{30}$$

and O_4 is the order of the time that takes to generate the rule (line [4]). We can see the number of elements to be written in the PLTL program is clearly dominated by $|\mathcal{S}_{tp}|$, which in turn is at most the maximum number of states defining a trap, less or equal than n and then

$$O_4 = n \tag{31}$$

From eqs. 29, 30 and 31 we have that $O_i = n^2$ (eq. 28).
From eqs. 27 and 28 we have that $O_{step4} = n^3$ (eq. 26). △

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

Sketch of proof **5** Clearly, the order of step 5 is dominated by the generation time (line [1]) which in turn is proportional to the number of processes in the PARADIGMmodel. As this number is less or equal than n, step 5 is $O(n)$. △

Theorems 1, 2, 3, 4 and 5 support our claim, i.e., the entire translation algorithm runs in polynomial time. In fact, it is at most $O(n^5)$.

6. AN EXAMPLE

Next we show a PLTL program generated by the translation of the partial specification of the ATM case study. Also remember although manager ATM is also involved with subprocess checkPIN, we are just focusing on manager BankComputer to illustrate the algorithm. We remind the reader a full version of the example and the corresponding set of rules is available in [2].

```
      % STATE CHANGES IN EMPLOYEE PROCESSES

% in checkPIN()


□((cpChecked ∧ cPs1) →
          ◇(¬ cpChecked ∧ cpNotChecking))
□((cpNotChecking ∧ cPs2) →
          ◇(¬ cpNotChecking ∧ cpConnected))
□((cpVerifying ∧ cPs3) →
          ◇(¬ cpVerifying ∧ cpChecked))
□((cpConnected ∧ cPs4) →
          ◇(¬ cpConnected ∧ cpVerifying))
```

```
% in verifyAccount()

□((vaAccountOK ∧ vAs1) →
          ◇(¬ vaAccountOK ∧ vaNotVerifying))
□((vaAccountNotOK ∧ vAs1) →
          ◇(¬ vaAccountNotOK ∧ vaNotVerifying))
□((vaNotVerifying ∧ vAs2) →
          ◇(¬ vaNotVerifying ∧ vaEncrypted))
□((vaEncrypted ∧ vAs2) →
          ◇( ¬ vaEncrypted
             ∧
             (vaAccountOK ⊕ vaAccountNotOK)))



          % SUBPROCESS PRESCRIPTIONS


% by manager BankComputer


□(bcWaiting → (cPs4 ∧ gMs4 ∧ pTs1 ∧ vAs1))
□(bcVerifying → (cPs4 ∧ gMs4 ∧ pTs1 ∧ vAs2))



      % STATE CHANGES IN MANAGER PROCESSES


% in BankComputer

□((bcWaiting ∧ tcP4 ∧ tvA1 ) →
          ◇ (¬ bcWaiting ∧ bcVerifying ∧
             ¬ vAs1 ∧ ¬ tvA1))



               % INSIDE TRAPS


% belonging to checkPIN()


□((cPs1 ∧ cpNotChecking) → tcP1)
□((cPs2 ∧
   (cpConnected ∨ cpVerifying ∨ cpChecked)
                                    → tcP2)
□((cPs3 ∧
   (cpNotChecking ∨ cpConnected ∨ cpChecked))
                                    → tcP3)
□((cPs4 ∧ cpVerifying) → tcP4)


% belonging to verifyAccount()


□((vAs1 ∧ vaNotVerifying) → tvA1)
□((vAs2 ∧ vaAccountOK) → tvA2)
□((vAs2 ∧ vaAccountNotOK) → tvA3)
```

```
                % INITIAL CONDITIONS


init

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

7. MODEL VERIFICATION

This section shows that is possible to link the PLTL program resulting from the translation, to a verification procedure about correctness in the initial PARADIGM model. We show that well-known properties in the system verification literature [12] can be naturally expressed in the PLTL program. Further research is needed to link these notions to the already available tools SPIN and STeP.

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

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

EXAMPLE **7** (*guarantee* property) *"It is possible for the ATM to report a PIN as checked while BankComputer is still verifying it."*

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

EXAMPLE **8** (*response* properties) *"All PIN verifications will eventually end"*

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

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

$$\Box(\texttt{tcP4} \rightarrow \Diamond(\texttt{tvA2} \vee \texttt{tvA3}))$$

$$\triangle$$

EXAMPLE **9** (*response/recurrence* property) *"The stage of verifying account implies to check if the account is acceptable or not. After that step the process is re-started."*

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

$$\triangle$$

EXAMPLE **10** A *recurrence* property: *"The process of checking a PIN can be cyclically invoked"*

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

However, some readers may find the following two rules easier to understand:

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

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

It can be seen that the verification process can be set, either at the more general level of the functionality of the system (examples 6 and 9) 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 to verify temporal properties. Other alternatives includes 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, providing better scalability than tools centered on state-exploration like SPIN.

As our work addresses some temporal aspects implicit in PARADIGM specifications, it may help to encode these notions in Fair Transition Systems, in the case of STeP, or global automata, in the case of SPIN. We also believe that our specification language may be linked to other verification frameworks. Indeed, Etessami [9] discusses a translation of an extended version of Linear Temporal Logic (LTL) to Büchi Automata.

8. CONCLUSIONS

We have introduced a translation process that takes a PARADIGM model and generates a PLTL program which expresses, from a declarative approach, the dynamic behavior of described by the model. This program can be used to trace process interactions and verify, to a certain extent, correctness properties. For example, classical properties such as guarantee, persistence, response and others can be queried to verify correctness of PARADIGM models.

We focused on a polynomial translation algorithm (a Prolog implementation is discussed in [2]). We complement a previous conference paper ([3]) which introduced the main concepts behind the translation process and its application in system verification. This work is certainly worth to be compared with [10], where a transition-like operational semantics is considered. However algorithms and verification possibilities are not addressed in depth.

Further work is needed to link PARADIGM with verification tools like Step and SPIN, but nevertheless we have revealed some important insights on the dynamic aspects of PARADIGM models. This will hopefully encourage further research on verification of PARADIGM-modelled systems.

9. REFERENCES

[1] T. Arai and F. Stolzenburg. Multiagent Systems Specification by UML Statecharts Aiming at Intelligent Manufacturing. Technical report, Universität Koblenz-Landau, December, 2001.

[2] J. C. Augusto and R. S. Gómez. A Procedure to Translate Paradigm Specifications to Propositional Linear Temporal Logic and its Application to Verification. Technical report, Department of Electronics and Computer Science, University of Southampton, U.K., 2002. 42 pages, available at `http://www.ecs.soton.ac.uk/~jca/Par2PLTL.pdf`.

[3] Juan C. Augusto and Rodolfo S. Gómez. A temporal logic view of paradigm specifications. In*Proceedings of Fourteenth International Conference on Software Engineering and Knowledge Engineering (SEKE02)*, pages 497–503, Ischia, Italy, July 2002.

[4] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, B. Sipma, and T. Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design*, 1999.

[5] G. Brassard and P. Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996.

[6] M. L. Cobo and J. C. Augusto. Logical Foundations and Implementation of an Extension of Temporal Prolog. *The Journal of Computer Science and Technology (JCS&T), sponsored by ISTEC (Iberoamerican Science & Technology Education Consortium)*, 1(2):22–36, 1999.

[7] T. de Buntje, G. Engels, L. Groenewegen, and A. Matsinger. Industrial Maintenance Modelled in SOCCA. InRijn-beek, editor, *Fourth International Conference on the Software Process. Proceedings*, pages 13–26. IEEE Computer Society Press, 1996.

[8] J. Ebert, L. Groenewegen, and R. Süttenbach. A Formalization of SOCCA. Technical Report 10-99, Universität Koblenz-Landau, 1999.

[9] K. Etessami. Stutter-invariant languages, omega-automata, and temporal logic. In*Proceedings of 11th International Conference on Computer-Aided Verification (CAV'99)*, pages 236–248, 1999.

[10] L. Groenewegen and E. de Vink. Operational Semantics for Coordination in Paradigm. InFarhad Arbab and Carolyn L. Talcott, editor, *Coordination Models and Languages, 5th International Conference, COORDINATION 2002, YORK, UK, April 8-11, 2002, Proceedings*, volume 2315 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2002.

[11] G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[12] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems (Specification)*. Springer Verlag, 1992.

[13] A. Pnueli. Deduction is forever. An invited talk. In*Formal Methods'99*, Toulouse, France, September, 1999.

[14] M. van Steen, L. Groenewegen, and G. Oosting. Parallel Control Processes: Modular Parallelism and Communication. InHertzberger and Groen, editors, *Proceedings Intelligent Autonomous Systems*, pages 562–579, Amsterdam, The Netherlands, 1987.

[15] A. Wulms. Blackboard Systems Modelled in SOCCA. Master's thesis, Universität at Koblenz-Landau, 1997.

[16] S. Yovine. Kronos: A Verification Tool for Real-Time Systems. *Springer International Journal of Software Tools for Technology Transfer*, 1997.