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
United Kingdom
jca@ecs.soton.ac.uk

Rodolfo Sabás Gómez
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Argentina
rgomez@cs.uns.edu.ar

Abstract

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 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 useful on verifying 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.

Key Words: PARADIGM, Temporal Logic, Verification.
## Contents

1 Introduction 4

2 PARADIGM 5

3 The Temporal Logic 5

4 The translation process, conceptually 6
   4.1 State changes in employee processes 13
   4.2 Subprocess prescription 16
   4.3 State changes in manager processes 17
   4.4 Inside a trap 18
   4.5 Initial conditions 18

5 The translation process as an algorithm 19
   5.1 Input sets 19
   5.2 Steps 25
   5.3 Complexity 28

6 An example 34

7 Model verification 39

8 Conclusions and Further Work 40

9 Acknowledgments 41
## List of Figures

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

PARADIGM [vSGO87] is a high-level modelling language which has been proposed for designing parallel and cooperative systems. It is well known as being the sublanguage of SOCCA [EGS99] used for modelling object communication, coordination and cooperation. Basically, PARADIGM models a system as a set of parallel, communicating processes.

Propositional Linear Temporal Logic (PLTL) has been used in the specification of dynamic systems and verification of their behaviour correctness ([MP89] and [Pnu99]). Different specification and verification systems have been proposed in the literature, notably STeP [BBC+99] and SPIN [Hol97]. In the STeP framework SPL can be used to specify a system that is translated to a Fair Transition System. Then, behaviour 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 [Yov97], 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 specification to a PLTL-based program, thus obtaining an executable model for 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.

A number of benefits can be obtained from such a translation. Firstly, a temporal logic framework supports definition and automatic verification of certain desirable properties about the model. 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, [CA99] that provides interpretation for a subset of PLTL covering more of the properties discussed at the end of this article. Secondly, 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 behaviors until functional system requirements have been met. Finally, the logic approach offers a different, declarative way for studying PARADIGM models. We think this new features 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 paper is organized as follows. Section 2 explains the main concepts of PARADIGM models. Section 3 explains the logic framework 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. A complete example is developed in section 6. Section 7 shows some examples of verification properties. 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 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 a 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 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, for an employee could perform a given transition, this transition must be contained in all of its currently prescribed subprocesses. For the sake of simplicity, we have assumed all processes of the PARADIGM model are always active.

Traps model the points in execution where employees need coordination. They are defined as being a subset of subprocess states. Once a 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 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 it is possible for a subprocess to be prescribed on 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 these traps must be entered for the transition could be performed. Employee executions cannot proceed outside of traps until the manager prescribes the proper set of subprocesses, thus changing their behavior 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 [EGS99].

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 [CA99].

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 relation ≤. The system being specified then will 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 \( \theta \) 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 $L_P$ 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). For simplification 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:

$$\phi = p | \neg \phi | \phi_1 \land \phi_2 | \phi_1 \lor \phi_2 | \phi_1 \rightarrow \phi_2 | \Diamond \phi | \Box \phi$$

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

$$(\sigma, t) |\!\!\!| = p \iff p \in s_t \text{ with } p \text{ atomic}$$

$$(\sigma, t) |\!\!\!| = \neg \phi_1 \iff (\sigma, t) \not|\!\!\!| = \phi_1$$

$$(\sigma, t) |\!\!\!| = \phi_1 \lor \phi_2 \iff (\sigma, t) |\!\!\!| = \phi_1 \text{ or } (\sigma, t) |\!\!\!| = \phi_2$$

$$(\sigma, t) |\!\!\!| = \phi_1 \land \phi_2 \iff (\sigma, t) |\!\!\!| = \phi_1 \text{ and } (\sigma, t) |\!\!\!| = \phi_2$$

$$(\sigma, t) |\!\!\!| = \phi_1 \rightarrow \phi_2 \iff (\sigma, t) \not|\!\!\!| = \phi_1 \text{ or } (\sigma, t) |\!\!\!| = \phi_2$$

$$(\sigma, t) |\!\!\!| = \Diamond \phi \iff \text{there exists } s > t : (\sigma, s) |\!\!\!| = \phi$$

$$(\sigma, t) |\!\!\!| = \Box \phi \iff \text{for all } s > t : (\sigma, s) |\!\!\!| = \phi$$

This language will give us a set of well formed formulas that is rich enough to encode in a declarative way the PARADIGM specification. It also allow us to represent well known schema formulas [MP92] that can be used to query the resulting temporal logic program in order to verify correctness of behaviour. Some examples of this formulas are: $\Box \phi$ (safety) and others from the “liveness family” like $\Diamond \phi$ (guarantee), $\mathcal{U}(\phi_1 \rightarrow \Diamond \phi_2)$ (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 is dependent on the sets of manager and employee processes, they should not be prohibitively large sets as modularity will demand to keep manager and employee processes sets reasonably small.

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

4 The translation process, conceptually

The goal of the translation process is to produce a PLTL-based program, $\mathcal{P}$, which simulates the behavior of the processes included in the PARADIGM specification. This work focuses on the elements of a PARADIGM specification that are translated, and the logic rules that result as an outcome. Although we will not give too specific details about how the translation is actually performed (section 5), the main steps of the translation procedure are explained and exemplified.
The evolution of process executions can be expressed as a sequence of time periods: a) the time processes remain on each state, b) the time subprocesses remain prescribed to each employee and c) the time employees remain inside their traps. This knowledge will be expressed by a set of propositions: 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 will assume all propositions denoting states, subprocesses and traps are unique.

Program $P$ will be constructed as a collection of rules implying the validity periods for propositions $ST$, $SP$ and $TP$. Rules will assert or deny the truth of these propositions at a given time, depending on the set of preconditions that are true at that time. These assertions and denials, together with the persistence semantics of our logic framework, are enough to model the time periods where propositions are true. These rules will be introduced in section 4.1, 4.2, 4.3, 4.4 and 4.5.

It is worth mentioning that state changes can take any amount of time to be performed. This include the time processes remain on their states and the time transitions take to be performed. As this information is not provided by PARADIGM models, we have respected its ordering semantics: we only reflect the order in which states can be visited, not the time it takes. Thus, state changes will be expressed by rules with schema $\Box (Pre \rightarrow \Diamond Pos)$ where $Pre$ is a set of preconditions which must hold for the change could be performed, and $Pos$ is a set of postconditions holding after the change. Note the use of $\Diamond$, expressing that the change eventually occurs, but we cannot be sure when it does.

Translation steps will be better explained through an example we have adapted from [EGS99]. In [EGS99] the ATM system is modelled in SOCCA. Strictly speaking, only Figs. 3 to 20 describe the PARADIGM model because it is just one of the four perspectives which are used in SOCCA for modelling a system. Nevertheless, we have decided to show other perspectives for making 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 set 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 function.

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 user’s magnetic card with the personal identification number, but for doing 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.

---

**Figure 1: Data perspective**
class ATM

Figure 2: Behavior perspective

class BankComputer

Figure 3: Employee process checkPIN

Checking
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 \texttt{processTransaction}

Figure 12: Subprocesses of \texttt{processTransaction}

Figure 13: Manager process \texttt{BankComputer}
Next we introduce the rules composing program $P$.

### 4.1 State changes in employee processes

This kind of rules implies the time each employee remains on a given state. Let $ts$ be a transition from state $ST_i$ to state $ST_j$ in a given employee $E$. For this state change could be performed at time $t$ 

(a) $E$ must be currently on $ST_i$ and 

(b) all subprocesses that are currently prescribed to $E$ must contain $ts$. Precondition (a) can be expressed by requesting proposition $ST_i$ to be valid at $t$. Precondition (b) deserves a deeper explanation.

Let $M_E = \{M_1, \ldots, M_q\}$ be the set of all managers for $E$. Of all subprocesses that can be
Figure 17: Subprocesses of ejectCard

Figure 18: Employee process cancel

Figure 19: Subprocesses of cancel
Figure 20: Manager process ATM
prescribed to \( E \) by manager \( M_i \in M_E \), let \( S_i = \{ SP_1^i, \ldots, SP_n^i \} \) be the set of those which contains \( t_s \). For each manager \( M_i \) it is requested at least one subprocess included in \( S_i \) to be prescribed to \( E \) at time \( t \). This precondition is expressed by the following conjunction:

\[
((SP_1^i \lor \cdots \lor SP_r^i) \land \cdots \land (SP_q^i \lor \cdots \lor SP_s^i))
\]

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:

\[
\Box((ST_i \land (SP_1^i \lor \cdots \lor SP_r^i) \land \cdots \land (SP_q^i \lor \cdots \lor SP_s^i)) \rightarrow \Diamond(\neg ST_i \land ST_j))
\]

Example 1 shows a state change from Connected to Processed in employee \( \text{getMoney} \). Fig. 7 shows that transition “call PT” is allowed in both subprocesses which can be prescribed by \( \text{ATM} \), \( \text{getMoney}_s1 \) and \( \text{getMoney}_s2 \). But Fig. 8 shows that \( \text{getMoney}_s4 \) is the only subprocess, of those which can be prescribed by \( \text{BankComputer} \), that contains “call PT”. Thus, it does not matter which subprocess is \( \text{ATM} \) currently prescribing, \( \text{getMoney}_s1 \) or \( \text{getMoney}_s2 \) but \( \text{BankComputer} \) must be prescribing \( \text{getMoney}_s4 \). Otherwise, i.e. if \( \text{getMoney}_s3 \) is currently prescribed, the state change cannot be performed. This is expressed in the rule as \((gMs1 \lor gMs2) \land gMs4)\).

**Example 1**

\[
\Box((\neg \text{cpConnected} \land (gMs1 \lor gMs2) \land gMs4) \rightarrow \Diamond(\neg \text{cpConnected} \land \text{cpProcess}))
\]

Note this rule admmits some kind of optimization: it can be proved that disjunction \((gMs1 \lor gMs2)\) is not really needed. Manager \( \text{ATM} \) is always prescribing a subprocess to \( \text{getMoney} \), be it \( \text{getMoney}_s1 \) or \( \text{getMoney}_s2 \), thus it will always be the case that one of these two subprocesses will be prescribed by the time \( \text{getMoney} \) tries to change from state Connected to Processed. As a consequence, disjunction \((gMs1 \lor gMs2)\) is always true. In other words, this optimization can be done when all subprocesses that can be prescribed by a given manager contain a given transition. In our example, both \( \text{getMoney}_s1 \) and \( \text{getMoney}_s2 \) contain transition “call PT”, i.e. manager \( \text{ATM} \) can never impose a restriction on \( \text{getMoney} \) performing the change from Connected to Processed. The rule above can then be re-written as follows:

\[
\Box((\neg \text{cpConnected} \land gMs4) \rightarrow \Diamond(\neg \text{cpConnected} \land \text{cpVerifying}))
\]

This optimization strategy is reflected in the algorithm of section 5, step 1.

4.2 Subprocess prescription

This kind of rules expresses 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 that are prescribed while the manager remains on that state. Let \( ST_i \) be a manager state and \( S_i = \{ SP_1, \ldots, SP_n \} \) be the set of subprocesses prescribed in this state. The translation process will generate the following rule:
Example 2 describes the set of subprocesses that \texttt{BankComputer} prescribes in states \texttt{Waiting} and \texttt{Verifying} (Fig. 13). Those whose labels have the prefixes \texttt{cP} and \texttt{vA} denotes subprocesses of \texttt{checkPIN} (Figs. 4 and 5) and \texttt{verifyAccount} (Fig. 10), respectively. These labels are easy to understand, for example \texttt{cP}s\texttt{4} denotes subprocess \texttt{checkPIN}s\texttt{4}. We can also see that other subprocesses are prescribed, those whose labels have the prefixes \texttt{gM} and \texttt{pT} are subprocesses of \texttt{getMoney} (Figs. 7 and 8) and \texttt{processTransaction} (Fig. 12), respectively.

\textbf{Example 2}

\[\Box(\text{bcWaiting} \implies \text{cP}s\texttt{4} \land \text{gM}s\texttt{4} \land \text{vA}s\texttt{1} \land \text{pT}s\texttt{1})]\
\[\Box(\text{bcVerifying} \implies \text{cP}s\texttt{4} \land \text{gM}s\texttt{4} \land \text{vA}s\texttt{2} \land \text{pT}s\texttt{1})]\

\[\triangle\]

\subsection{4.3 State changes in manager processes}

This kind of rules implies the time each manager remains on a given state. According to PAR-ADIGM model semantics, these rules also imply the time when subprocesses and traps are left. Let \( ts \) be a transition from state \( ST_i \) to state \( ST_j \) in a given manager \( M \). For this state change could be performed at time \( t \) \textbf{a)} \( M \) must be currently on \( ST_i \) and \textbf{b)} the proper employees must be currently inside those traps related to \( ts \) (see section 2). Precondition (\( a \)) can be expressed by requesting proposition \( ST_i \) to be valid at \( t \). Let \( T_{\text{entered}} = \{TP_1, \ldots, TP_n\} \) be the set of traps related to transition \( ts \). Precondition (\( b \)) can be expressed by requesting proposition \( TP_i \) to be valid at \( t \), for all \( TP_i \in T_{\text{entered}} \).

After the state change, i.e. at time \( t + n \), \textbf{a)} \( M \) will be no longer on \( ST_i \) but on \( ST_j \) and \textbf{b)} it is possible for some of those subprocesses prescribed in \( ST_i \) not to be prescribed in \( ST_j \). As a consequence all traps belonging to those subprocesses will be left. Postcondition (\( a \)) can be expressed by asserting propositions \( \neg ST_i \) and \( ST_j \). Let \( S_{\text{left}} = \{SP_1, \ldots, SP_m\} \) be the set of subprocesses prescribed in \( ST_i \) but not in \( ST_j \), and \( T_{\text{left}} = \{TP_{\text{q}}, \ldots, TP_u\} \) the set of traps included in subprocesses of \( S_{\text{left}} \). Postcondition (\( b \)) can be expressed by asserting \( \neg SP_i \), for all \( SP_i \in S_{\text{left}} \), and \( \neg TP_i \), for all \( TP_i \in T_{\text{left}} \). Finally, those subprocesses of \( ST_i \) that 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 describing subprocess prescriptions in \( ST_j \) (see section 4.2). Rules modelling state changes in manager processes have the following schema:

\[\Box((ST_i \land (TP_1 \land \ldots \land TP_n))\rightarrow \Diamond(\neg ST_i \land ST_j \land (\neg SP_1 \land \ldots \land \neg SP_m) \land (\neg TP_{\text{q}} \land \ldots \land \neg TP_u)))\]

Example 3 describes the state change from state \texttt{Waiting} to state \texttt{Verifying} in manager \texttt{BankComputer} (Fig. 13). This change cannot be performed until both traps \( T-\text{cP}4 \) and \( T-\text{vA}1 \) have been entered. As in state \texttt{Waiting} the manager is prescribing subprocesses \texttt{checkPIN}s\texttt{4} and \texttt{verifyAccount}s\texttt{1}, this means for the manager could change to state \texttt{Verifying} \textbf{a)} \texttt{checkPIN}
must be in state Connected, i.e. it should have called verifyAccount (see checkPIN_s4 in Fig. 5), and b) verifyAccount must be in state NotVerifying, i.e. prepared 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 the fact 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 checkPIN cannot leave trap T-cP4. This means checkPIN cannot proceed until verifyAccount ends its job. Proposition bcVerifying and the rule shown in example 2 express the fact a new subprocess, verifyAccount_s2, is now prescribed and that checkPIN_s4 remains prescribed.

EXAMPLE 3 State change from state Waiting to state Verifying in manager BankComputer

□((bcWaiting ∧ tcP4 ∧ tvA1) → ◊(¬bcWaiting ∧ bcVerifying ∧ ¬vAs1 ∧ ¬tvA1)) △

4.4 Inside a trap

This kind of rules implies 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. Note this information is needed by the rules which express state changes in manager processes (see section 4.3 above).

Let ST_1, ..., 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 ∈ ST_P. Thus, the translation will generate rules with the following schema:

□((SP ∧ (ST_1 ∨ ... ∨ ST_n)) → TP)

Example 4 expresses the fact employee checkPIN remains inside trap T-cP2 as long as it is prescribed subprocess checkPIN_s2 and it remains on states Connected, Verifying or Checked.

EXAMPLE 4

□((cPs2 ∧ (cpConnected ∨ cpVerifying ∨ cpChecked)) → tcP2) △

4.5 Initial conditions

All processes are supposed to start their executions coordinately. Of course, subprocesses that 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 that only holds at the initial time, and ST_1, ..., ST_n the set of initial states of all processes. The translation will generate rules with the following schema:

init
init → (ST_1 ∧ ... ∧ ST_n)
Example 5 shows 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 the first subprocesses to be prescribed by BankComputer to checkPIN and verifyAccount are, respectively, checkPIN_s4 and verifyAccount_s1 (see Fig. 13 and Example 2).

**Example 5** Initial states

\[
\text{init} \\
\text{init} \rightarrow ( \text{cpNotChecking} \land \text{vaNotVerifying} \land \text{bcWaiting} )
\]

Although for simplicity we have supposed that initial conditions are generated by the translation process, it is perfectly possible for this information to be supplied by the user. She/he could specify different sets of initial states for every process, thus obtaining a different simulation for the system behavior.

Finally, we finish this section by warning the reader that program \( P \) is defined as the union of all rules (sections 4.1, 4.2, 4.3, 4.4 and 4.5) generated so far by the translation process.

## 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. The PARADIGM specification is assumed to be correct, and it contains all the information needed for the algorithm to produce the PLTL program. As a matter of true, 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 include 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 for 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 for constructing these sets. We have chosen a set of labels for denoting 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.
1. A finite set \( EMP \) denoting all employee processes. In the example (Figs. 3, 6, 9 and 11) we have:
\[
EMP = \{ \text{checkPIN, getMoney, verifyAccount, processTransaction, readCard, ejectCard, cancel} \}.
\]

2. A finite set \( MAN \) denoting all manager processes. In the example (Figs. 20 and 13) we have:
\[
MAN = \{ \text{atm, bankComputer} \}.
\]

3. A finite set \( PRO_{\text{transitions}} \) denoting the set of transitions of every process.
\[
PRO_{\text{transitions}} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} \{(ST_j, ST_k)\}
\]
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, 14, 16, 18, 20 and 13) we have:

\[
\begin{align*}
(\text{checkPIN}, \{(\text{cpNotChecking, cpConnected}),
&\quad (\text{cpConnected, cpVerifying}),
&\quad (\text{cpVerifying, cpCheckOK}),
&\quad (\text{cpVerifying, cpCheckNotOK}),
&\quad (\text{cpCheckOK, cpNotChecking}),
&\quad (\text{cpCheckNotOK, cpNotChecking})\}), \\
(\text{getMoney}, \{(\text{gmNotGetting, gmConnected}),
&\quad (\text{gmConnected, gmProcessed}),
&\quad (\text{gmProcessed, gmMoneyPaid}),
&\quad (\text{gmMoneyPaid, gmNotGetting})\}), \\
(\text{verifyAccount}, \{(\text{vaNotVerifying, vaEncrypted}),
&\quad (\text{vaEncrypted, vaAccountVerifiedOK}),
&\quad (\text{vaEncrypted, vaAccountVerifiedNotOK}),
&\quad (\text{vaAccountVerifiedOK, vaNotVerifying}),
&\quad (\text{vaAccountVerifiedNotOK, vaNotVerifying})\}), \\
(\text{processTransaction}, \{(\text{ptNotProcessing, ptWaitingForProcessing}),
&\quad (\text{ptWaitingForProcessing, ptProcessed}),
&\quad (\text{ptProcessed, ptNotProcessing})\}), \\
(\text{readCard}, \{(\text{rcNotReading, rcReading}),
&\quad (\text{rcReading, rcCardOK}),
&\quad (\text{rcReading, rcCardNotOK}),
&\quad (\text{rcCardOK, rcNotReading})\}), \\
(\text{ejectCard}, \{(\text{ecNotEjecting, ecWillingToEject}),
&\quad (\text{ecWillingToEject, ecEjected}),
&\quad (\text{ecEjected, ecNotEjecting})\}) \\
(\text{cancel}, \{(\text{caNot Cancelling, caWillingToCancel}),
&\quad (\text{caWillingToCancel, caCancelled}),
&\quad (\text{caCancelled, caNot Cancelling})\}) \\
(\text{atm}, \{(\text{atmWaiting, atmReadingCard}),
&\quad (\text{atmReadingCard, atmChekingPIN})\}
\end{align*}
\]
\[
\{(\text{atmChekingPIN}, \text{atmPINChecked}), \\
(\text{atmPINChecked}, \text{atmGettingMoney}), \\
(\text{atmGettingMoney}, \text{atmPayingOff}), \\
(\text{atmPayingOff}, \text{atmCardEjected}), \\
(\text{atmCardEjected}, \text{atmWaiting}), \\
(\text{atmReadingCard}, \text{atmNotOK}), \\
(\text{atmChekingPIN}, \text{atmNotOK}), \\
(\text{atmReadingCard}, \text{atmCancelling}), \\
(\text{atmChekingPIN}, \text{atmCancelling}), \\
(\text{atmNotOK}, \text{atmCancelling}), \\
(\text{atmCancelling}, \text{atmCancelled}), \\
(\text{atmCancelled}, \text{atmCardEjected})\}
\]

\{(\text{bankComputer}, \{\text{bcWaiting}, \text{bcVerifying}\}), \\
(\text{bcVerifying}, \text{bcAccountVerifiedOK}), \\
(\text{bcAccountVerifiedOK}, \text{bcWaitingForTransactionRequest}), \\
(\text{bcWaitingForTransactionRequest}, \text{bcProcessing}), \\
(\text{bcProcessing}, \text{bcTransactionProcessed}), \\
(\text{bcTransactionProcessed}, \text{bcWaiting}), \\
(\text{bcVerifying}, \text{bcAccountVerifiedNotOK}), \\
(\text{bcAccountVerifiedNotOK}, \text{bcWaiting})\}

4. A finite set \( MAN\) _subprocesses_ denoting the set of subprocess prescribed in every manager state. \( MAN\) _subprocesses_ = \( \bigcup^n_{i=1} \{(ST_i, \bigcup^m_{j=1} \{SP_j\})\}\) where \( ST_i \) denotes a manager state and \( SP_j \) denotes a subprocess prescribed in \( ST_i \). In the example (Figs. 20 and 13) we have:

\[
MAN\text{subprocesses} = \{
\{(\text{atmWaiting}, \{\text{cPs1}, \text{gMs1}, \text{eCs1}, \text{rCs1}, \text{cAs1}\}), \\
(\text{atmReadingCard}, \{\text{cPs1}, \text{gMs1}, \text{eCs1}, \text{rCs2}, \text{cAs1}\}), \\
(\text{atmChekingPIN}, \{\text{cPs2}, \text{gMs1}, \text{eCs1}, \text{rCs1}, \text{cAs1}\}), \\
(\text{atmPINChecked}, \{\text{cPs1}, \text{gMs1}, \text{eCs1}, \text{rCs1}, \text{cAs1}\}), \\
(\text{atmGettingMoney}, \{\text{cPs1}, \text{gMs2}, \text{eCs1}, \text{rCs1}, \text{cAs1}\}), \\
(\text{atmPayingOff}, \{\text{cPs1}, \text{gMs1}, \text{eCs1}, \text{rCs1}, \text{cAs1}\}), \\
(\text{atmCardEjected}, \{\text{cPs1}, \text{gMs1}, \text{eCs2}, \text{rCs1}, \text{cAs1}\}), \\
(\text{atmNotOK}, \{\text{cPs1}, \text{gMs1}, \text{eCs1}, \text{rCs1}, \text{cAs1}\}), \\
(\text{atmCancelling}, \{\text{cPs1}, \text{gMs1}, \text{eCs1}, \text{rCs1}, \text{cAs2}\}), \\
(\text{atmCancelled}, \{\text{cPs1}, \text{gMs1}, \text{eCs1}, \text{rCs1}, \text{cAs1}\}), \\
(\text{bcWaiting}, \{\text{cPs4}, \text{gMs4}, \text{pTs1}, \text{vAs1}\}), \\
(\text{bcVerifying}, \{\text{cPs4}, \text{gMs4}, \text{pTs1}, \text{vAs2}\}), \\
(\text{bcAccountVerifiedOK}, \{\text{cPs3}, \text{gMs4}, \text{pTs1}, \text{vAs1}\}), \\
(\text{bcWaitingForTransactionRequest}, \{\text{cPs4}, \text{gMs4}, \text{pTs1}, \text{vAs1}\}), \\
(\text{bcProcessing}, \{\text{cPs4}, \text{gMs4}, \text{pTs2}, \text{vAs1}\}), \\
(\text{bcTransactionProcessed}, \{\text{cPs4}, \text{gMs3}, \text{pTs1}, \text{vAs1}\}), \\
(\text{bcAccountVerifiedNotOK}, \{\text{cPs3}, \text{gMs4}, \text{pTs1}, \text{vAs1}\})\}
\]
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 trap \( TP_i \). In the example (Figs. 4, 5 and 3, 7, 8 and 6, 10 and 9, 12 and 11, 15 and 14, 17 and 16, and 19 and 18) we have:

\[
TRP_{states} = \{
\begin{align*}
(tcP1, \{ cpNotChecking \}), \\
(tcP2, \{ cpCheckOK \}), \\
(tcP3, \{ cpCheckNotOK \}), \\
(tcP4, \{ cpNotChecking, cpConnected, cpCheckOK, cpCheckNotOK \}), \\
(tcP5, \{ cpVerifying \}), \\
(tgM1, \{ gmNotGetting \}), \\
(tgM2, \{ gmConnected, gmProcessed, gmMoneyPaid \}), \\
(tgM3, \{ gmNotGetting, gmConnected \}), \\
(tgM4, \{ gmProcessed \}), \\
(tvA1, \{ vaNotVerifying \}), \\
(tvA2, \{ vaAccountVerifiedOK \}), \\
(tvA3, \{ vaAccountVerifiedNotOK \}), \\
(tpT1, \{ ptNotProcessing \}), \\
(tpT2, \{ ptProcessed \}), \\
(trC1, \{ rcReading \}), \\
(trC2, \{ rcCardOK \}), \\
(trC3, \{ rcCardNotOK \}), \\
teC1, \{ ecNotEjecting \}), \\
teC2, \{ ecWillingToEject, ecEjected \}), \\
tcA1, \{ caNotCancelling \}), \\
tcA2, \{ caWillingToCancel, caCancelled \})
\end{align*}
\]

6. A finite set \( SPR_{traps} \) denoting the set of traps of every subprocess.
\[
SPR_{traps} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} \{ (SP_i, TP_j) \}
\]
where \( SP_i \) denotes a subprocess and \( TP_j \) denotes a trap of \( SP_i \). In the example (Figs. 4, 5, 7, 8, 12, 15, 17 and 19) we have:

\[
SPR_{traps} = \{
\begin{align*}
(cPs1, \{ tcP1 \}), (cPs2, \{ tcP2, tcP3 \}), (cPs3, \{ tcP5 \}), (cPs4, \{ tcP5 \}), \\
(gMs1, \{ tgM1 \}), (gMs2, \{ tgM2 \}), (gMs3, \{ tgM3 \}), (gMs4, \{ tgM4 \}), \\
vAs1, \{ tvA1 \}), (vAs2, \{ tvA2, tvA3 \}), \\
(pTs1, \{ tpT1 \}), (pTs2, \{ tpT2 \}), \\
(rCs1, \{ trC1 \}), (rCs2, \{ trC2, trC3 \}), \\
(ecS1, \{ ecC1 \}), (ecS2, \{ ecC2 \}), \\
cAs1, \{ tcA1 \}), (cAs2, \{ tcA2 \})
\end{align*}
\]

7. A finite set \( EMP_{subprocesses} \) denoting, for every employee, the set of subprocesses which can be prescribed by every manager. \( EMP_{subprocesses} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} \{ (E_i, M_j, SP_k) \} \) 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:
\[ EMP_{\text{subprocesses}} = \{(\text{checkPIN, atm, \{cPs1, cPs2\}}), \]
\[ (\text{checkPIN, bankComputer,\{cPs3, cPs4\}}), \]
\[ (\text{getMoney, atm, \{gMs1, gMs2\}}), \]
\[ (\text{getMoney, bankComputer, \{gMs3, gMs4\}}), \]
\[ (\text{verifyAccount, bankComputer, \{vAs1, vas2\}}), \]
\[ (\text{processTransaction, bankComputer, \{pTs1, pTs2\}}), \]
\[ (\text{readCard, atm, \{rCs1, rCs2\}}), \]
\[ (\text{ejectCard, atm, \{eCs1, eCs2\}}), \]
\[ (\text{cancel, atm, \{cAs1, cAs2\}}) \}\]

8. A finite set \(INI_{\text{states}}\) denoting the initial state of every process. \(INI_{\text{states}} = \bigcup_{i=1}^{n} \{ST_i\}\) where \(ST_i\) denotes the initial state of a process. In the example (Figs. 3, 6, 9, 11, 14, 16, 18 and 13) we have:

\[ INI_{\text{states}} = \{ \text{cpNotChecking, gmNotGetting, vaNotVerifying, ptNotProcessing, rcNotReading, ecNotEjecting, caNotCancelling, atmWaiting, bcWaiting } \}\]

9. A finite set \(TRS_{\text{subprocesses}}\) denoting, for every employee transition, the set of subprocesses it is included in.
\[ TRS_{\text{subprocesses}} = \bigcup_{i=1}^{n} \{(ST_i, ST_j), \bigcup_{k=1}^{n} \{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:

\[ TRS_{\text{subprocesses}} = \{(cpNotChecking, cpConnected), \{cPs2, cPs3, cPs4\}), \]
\[ ((\text{cpConnected}, \text{cpVerifying}), \{cPs2, cPs4\}), \]
\[ ((\text{cpVerifying, cpCheckOK}), \{cPs2, cPs3\}), \]
\[ ((\text{cpVerifying, cpCheckNotOK}), \{cPs2, cPs3\}), \]
\[ ((\text{cpCheckOK, cpNotChecking}), \{cPs1, cPs3, cPs4\}), \]
\[ ((\text{cpCheckNotOK, cpNotChecking}), \{cPs1, cPs3, cPs4\}), \]
\[ ((\text{gmNotGetting, gmConnected}), \{gMs2, gMs3, gMs4\}), \]
\[ ((\text{gmConnected, gmProcessed}), \{gMs1, gMs2, gMs4\}), \]
\[ ((\text{gmProcessed, gmMoneyPaid}), \{gMs1, gMs2, gMs3\}), \]
\[ ((\text{gmMoneyPaid, gmNotGetting}), \{gMs1, gMs3\}), \]
\[ ((\text{vaNotVerifying, vaEncrypted}), \{vAs2\}), \]
\[ ((\text{vaEncrypted, vaAccountVerifiedOK}), \{vAs2\}), \]
\[ ((\text{vaEncrypted, vaAccountVerifiedNotOK}), \{vAs2\}), \]
\[ ((\text{vaAccountVerifiedOK, vaNotVerifying}), \{vAs1\}), \]
\[ ((\text{vaAccountVerifiedNotOK, vaNotVerifying}), \{vAs1\}), \]
\[ ((\text{ptNotProcessing, ptWaitingForProcessing}), \{pTs2\}), \]
\[ ((\text{ptWaitingForProcessing, ptProcessed}), \{pTs1, pTs2\}), \]
\[ ((\text{ptProcessed, ptNotProcessing}), \{pTs1\}), \]
\[ 23 \]
10. A finite set $\text{MAN}_{\text{traps}}$ denoting the set of traps that must be entered for every state change in a manager process could be performed. 

$$\text{MAN}_{\text{traps}} = \bigcup_{i=1}^{n} \{((ST_i, ST_j), \bigcup_{k=1}^{m} \{TP_k\})\}$$ 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. 20 and 13) we have:

$$\text{MAN}_{\text{traps}} = \{
\text{(atmWaiting, atmReadingCard), \{trC1\}},
\text{(atmReadingCard, atmCheckingPIN), \{trC2, tcP1\}},
\text{(atmCheckingPIN, atmPINChecked), \{tcP2\}},
\text{(atmPINChecked, atmGettingMoney), \{tcP1, tgM1\}},
\text{(atmGettingMoney, atmPayingOff), \{tgM2\}},
\text{(atmPayingOff, atmCardEjected), \{tgM1, teC1\}},
\text{(atmCardEjected, atmWaiting), \{teC2\}},
\text{(atmReadingCard, atmNotOK), \{trC3\}},
\text{(atmCheckingPIN, atmNotOK), \{tcP3\}},
\text{(atmReadingCard, atmCancelling), \{tcA1\}},
\text{(atmCheckingPIN, atmCancelling), \{tcA1\}},
\text{(atmNotOK, atmCancelling), \{tcA1\}},
\text{(atmCancelling, atmCancelled), \{tcA2\}},
\text{(atmCancelled, atmCardEjected), \{teC1\}},
\text{(bcWaiting, bcVerifying), \{tcP5, tvA1\}},
\text{(bcVerifying, bcAccountVerifiedOK), \{tvA2\}},
\text{(bcAccountVerifiedOK, bcWaitingForTransactionRequest), \{tcP4\}},
\text{(bcWaitingForTransactionRequest, bcProcessing), \{tgM4, tpT1\}},
\text{(bcProcessing, bcTransactionProcessed), \{tpT2\}},
\text{(bcTransactionProcessed, bcWaiting), \{tgM3\}},
\text{(bcVerifying, bcAccountVerifiedNotOK), \{tvA3\}},
\text{(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. $A$. 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 $[n]$. 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

$\text{Tmp1 := EMP ;}$

[1] Repeat until $\text{Tmp1} = \emptyset$

begin

Let $e \in \text{Tmp1}$ ;

$\text{Tmp1 := Tmp1/\{e\} ;}$

[2] Let $T_e$ such that $(e, T_e) \in PRO_{transitions}$

% for each transition of this employee

$\text{Tmp2 := T_e ;}$

[3] Repeat until $\text{Tmp2} = \emptyset$

begin

Let $(st_i, st_j) \in \text{Tmp2}$ ;

$\text{Tmp2 := Tmp2/(\{st_i, st_j\} ;}$

[4] Let $S_{ij}$ such that $((st_i, st_j), S_{ij}) \in TRS_{subprocesses}$

% $S_{ij}$ is the set of all subprocesses containing this transition

% $S_e$ is the set of all subprocesses prescribed by each manager

% to this employee

[5] Let $S_e = \{ S_M \mid \exists M \in MAN \ ( (e, M, S_M) \in EMP_{subprocesses} ) \}$ ;

% intersect each subset of $S_e$ with $S_{ij}$, and form the set $S_{ij}^M$

% Strict inclusion $T \subset S_M$ expresses the optimization

% described in section 4.1

[6] Let $S_{ij}^M = \{ T \mid \exists S_M \in S_e \ ( T = S_M \cap S_{ij} \ \land \ T \subset S_M ) \}$ ;

Suppose $S_{ij}^M = \{ \{sp_1^1, \ldots , sp_r^1\}, \ldots, \{sp_q^1, \ldots , sp_r^q\} \}$ ;

[7] GenerateRule($\square((st_i \land (sp_1^1 \lor \ldots \lor sp_r^1) \ \land \ldots \land (sp_1^q \lor \ldots \lor sp_r^q)) \rightarrow \Diamond (\neg st_i \land st_j)$)

end % {Repeat until Tmp2 = Ø}

end % {Repeat until Tmp1 = Ø}
2 : Subprocess prescriptions

INPUT: \( MAN_{\text{subprocesses}} \)

PROCEDURE:

\%
 for each manager state

\[ \text{Tmp1} := MAN_{\text{subprocesses}} ; \]

[1] Repeat until Tmp1 = \( \emptyset \)
begin

\%
\( S_{st} \) is the set of all subprocesses prescribed in this state

\( \text{Let} \ (st, S_{st}) \in \text{Tmp1} ; \)
\( \text{Tmp1 := Tmp1/\{(st, S_{st})\}} ; \)
\( \text{Suppose} \ S_{st} = \{sp_1, \ldots, sp_n\} ; \)

[2] \text{GenerateRule}( \Box(st \rightarrow (sp_1 \land \ldots \land sp_n)) )

end \%
{\{\text{Repeat until Tmp1 =} \emptyset\}}
3 : State changes in manager processes.

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

PROCEDURE:

% for each manager
Tmp1 := MAN ;

[1] Repeat until Tmp1 = Ø
begin
  Let m ∈ Tmp1
  Tmp1 := Tmp1/\{m\} ;

% \( T_m \) is the set of transitions of this manager

[2] Let \( T_m \) such that \((m, T_m) \in PRO_{transitions} ;
  Tmp2 := T_m ;

% for each transition of \( T_m \)

[3] Repeat until Tmp2 = Ø ;
begin
  Let \((s_i, s_j) \in Tmp2 ;
  Tmp2 := Tmp2/\{(s_i, s_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 \( T_{ij} \) such that \(((s_i, s_j), T_{ij}) \in MAN_{traps} ;

% \( I \) is the set of subprocesses prescribed in state \( s_i \)

[5] Let \( I \) such that \((s_i, I) \in MAN_{subprocesses} ;

% \( J \) is the set of subprocesses prescribed in state \( s_j \)

[6] Let \( J \) such that \((s_j, J) \in 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 \mid \exists SP \in D \ ( (SP, T_{SP}) \in SPR_{traps} \land TP \in T_{SP} ) \} ;
  Suppose \( T_{ij} = \{tp_1, ..., tp_n\} ;
  Suppose \( D = \{sp_1, ..., sp_m\} ;
  Suppose \( T_{left} = \{tp_q, ..., tp_u\} ;

[9] GenerateRule( \( \Box((s_i \land (tp_1 \land ... \land tp_n) \rightarrow \Diamond(\neg s_i \land s_j \land (\neg sp_1 \land ... \land \neg sp_m) \land (\neg tp_q \land ... \land \neg tp_u)))) )
  end % \{Repeat until Tmp2 = Ø\}
end % \{Repeat until Tmp1 = Ø\}
4 : Inside a trap.

INPUT: $SPR_{traps}, TRP_{states}$

PROCEDURE:

\% for each subprocess

$\text{Tmp1} := SPR_{traps}$ ;

[1] \Repeat until $\text{Tmp1} = \emptyset$
begin

\% $T$ is the set of traps of this subprocess

Let $(sp, T) \in \text{Tmp1}$ ;

$\text{Tmp1} := \text{Tmp1}/\{(sp, T)\}$ ;

\% for each trap in $T$

[2] \Repeat until $T = \emptyset$
begin

Let $tp \in T$ ;

$T := T/\{tp\}$ ;

\% $S_{tp}$ is the set of states defining this trap

[3] Let $S_{tp}$ such that $(tp, S_{tp}) \in TRP_{states}$ ;

Suppose $S_{tp} = \{st_1, \ldots, st_n\}$ ;

[4] GenerateRule( $\square(sp \land (st_1 \lor \ldots \lor st_n)) \rightarrow tp$ )
end
end

5 : Initial conditions.

INPUT: $INI_{states}$

PROCEDURE:

$\text{GenerateRule( init )}$ ;

Suppose $INI_{states} = \{st_1, \ldots, st_n\}$ ;

[1] $\text{GenerateRule( init } \rightarrow (st_1 \land \ldots \land st_n) \text{ )}$

5.3 Complexity

It can be proved that our translation algorithm runs in polynomial time. It is not our intention to offer a rigorous, formal proof of our claim but just give the reader an sketch of such a proof. Nevertheless we think it suffices to give the reader an idea of the algorithm efficiency.

We will develop our complexity analysis using the asymptotic notation often known as “the order of” or “big Oh” (see e.g. [BB96]). 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 $O(f(n)) = \{ g : \mathbb{N} \to \mathbb{R}^{\geq 0} \mid (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}) (\forall n \geq n_0)[g(n) \leq cf(n)] \}$.

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

Claim 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 execution time of our translation algorithm. Let $S_t, S_p$ and $T_p$ be the sets of all states, subprocesses and traps of $\mathcal{M}$, respectively. 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 $t(n) \in O(n^i)$, $i \in \mathbb{N}$, $n = max(|St|, |Sp|, |Tp|, |IS_1|, \ldots, |IS_m|)$. ■

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 some lines in the algorithm which has been marked with $[n]$. Function $\text{max}(a_1, \ldots, a_n)$ returns the maximum value among $a_1, \ldots, a_n$. $|S|$ denotes the cardinality of set $S$.

Theorem 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

$$O_{\text{step1}} = L_o \cdot \text{max}(O_2, O_3)$$

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

$$L_o = |Tmp1| = |EMP| \leq n$$

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

$$O_2 = n$$

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

$$O_3 = L_i \cdot \text{max}(O_4, O_5, O_6, O_7)$$

29
where $L_i$ is the number of iterations of the inner loop (line [3]), $L_i = |T_{mp2}| = |\mathcal{T}_e|$, 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 $S^M_{ij}$ (line [6]), which involves an intersection-inclusion proof for every element of set $\mathcal{S}_e$,

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

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 $|\mathcal{S}_m| \cdot |\mathcal{S}_{ij}|$. 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 $|\mathcal{I}| \cdot |\mathcal{S}_m|$. 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,

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

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 $|S^M_{ij}|$, which in turn is at most $|\mathcal{S}_e| \leq n$ and then

$$O_7 = n \quad (12)$$

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 $O_{step1} = n^5 \ (eq. \ 1)$. □

THEOREM 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

\[ O_{\text{step2}} = L_o.O_2 \]  

(13)

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

\[ L_o = |Tmp1| = |MAN_{\text{subprocesses}}| \leq n \]  

(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 \(-S_{st}--\), which in turn is at most the maximum number of subprocesses that a manager can prescribe on a single state, and then

\[ O_2 = n \]  

(15)

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

\[ \Box \]

Theorem 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

\[ O_{\text{step3}} = L_o.\max(O_2, O_3) \]  

(16)

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

\[ L_o = |Tmp1| = |MAN| \leq n \]  

(17)

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

\[ O_2 = n \]  

(18)

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

\[ O_3 = L_i.\max(O_4, O_5, O_6, O_7, O_8, O_9) \]  

(19)

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

\[ L_i \leq n \]  

(20)

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

\[ O_4 = n \]  

(21)

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

\[ O_5 = O_6 = n \]  

(22)

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

$$O_7 = n^2$$  \hspace{1cm} (23)

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

$$O_8 = n^2$$  \hspace{1cm} (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 $|T_{ij}| + |D| + |T_{left}|$. 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 PARADIGM model respectively. Therefore, the time of generation is at most $3n$ yielding

$$O_9 = n$$  \hspace{1cm} (25)

From eqs. 20, 21, 22, 23, 24 and 25 we have that $O_9 = n^3$ (eq. 19).

From eqs. 17 and 18 we have that $O_{step3} = n^4$ (eq. 16). $\square$

**THEOREM 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

$$O_{step4} = L_o.O_i$$  \hspace{1cm} (26)

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

$$L_o = |Tmp1| = |SPR_{traps}| \leq n$$  \hspace{1cm} (27)

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

$$O_i = L_i.max(O_3, O_4)$$  \hspace{1cm} (28)

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

$$L_i \leq n$$  \hspace{1cm} (29)

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

$$O_3 = n$$  \hspace{1cm} (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 $|S_{tp}|$, which in turn is at most the maximum number of states defining a trap, less or equal than $n$ and then

32
\[ O_4 = n \]  \quad (31)

From eqs. 29, 30 and 31 we have that \( O_4 = n^2 \) (eq. 28).

From eqs. 27 and 28 we have that \( O_{\text{step4}} = n^3 \) (eq. 26).

\[ \square \]

**Theorem 5** Rules expressing initial conditions (see step 5 in section 5.2) can be generated in \( O(n) \).

\[ \blacksquare \]

**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 PARADIGM model. As this number is less or equal than \( n \), step 5 is \( O(n) \).

\[ \square \]

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 complete a PTL program generated by the translation process. This program contains all the rules that are needed to simulate the ATM example processes.

%%%%%%%%%%%%%%%%%%%%%%%%%% Here begins the PTL program %%%%%%%%%%%%%%%%%%%%%%%%%%

% STATE CHANGES IN EMPLOYEE PROCESSES

% in checkPIN()

□((cpNotChecking ∧ cPs2) → ◊(¬ cpNotChecking ∧ cpConnected))
□((cpConnected ∧ cPs2 ∧ cPs4) → ◊(¬ cpConnected ∧ cpVerifying))
□((cpVerifying ∧ cPs2 ∧ cPs3) → ◊(¬ cpVerifying ∧ cpCheckOK))
□((cpVerifying ∧ cPs2 ∧ cPs3) → ◊(¬ cpVerifying ∧ cpCheckNotOK))
□((cpCheckOK ∧ cPs1) → ◊(¬ cpCheckOK ∧ cpNotChecking))
□((cpCheckNotOK ∧ cPs1) → ◊(¬ cpCheckNotOK ∧ cpNotChecking))

% in getMoney()

□((gmNotGetting ∧ gMs2) → ◊(¬ gmNotGetting ∧ gmConnected))
□((gmConnected ∧ gMs4) → ◊(¬ gmConnected ∧ gmProcessed))
□(gmProcessed ∧ gMs3) → ◊(¬ gmProcessed ∧ gmMoney Paid))
□((gmMoneyPaid ∧ gMs1 ∧ gMs3) → ◊(¬ gmMoneyPaid ∧ gmNotGetting))

% in verifyAccount()

□((vaNotVerifying ∧ vAs2) → ◊(¬ vaNotVerifying ∧ vaEncrypted))
□((vaEncrypted ∧ vAs2) → ◊(¬ vaEncrypted ∧ vaAccountVerifiedOK))
□((vaEncrypted ∧ vAs2) → ◊(¬ vaEncrypted ∧ vaAccountVerifiedNotOK))
□((vaAccountVerifiedOK ∧ vAs1) → ◊(¬ vaAccountVerifiedOK ∧ vaNotVerifying))
□((vaAccountVerifiedNotOK ∧ vAs1) → ◊(¬ vaAccountVerifiedNotOK ∧ vaNotVerifying))

% in processTransaction()

□((ptNotProcessing ∧ pTs2) → ◊(¬ ptNotProcessing ∧ ptWaitingForProcessing))
□(ptWaitingForProcessing → ◊(¬ ptWaitingForProcessing ∧ ptProcessed))
□((ptProcessed ∧ pTs1) → ◊(¬ ptProcessed ∧ ptNotProcessing))
% in readCard()

\[ \square((rcNotReading \land rCs1) \rightarrow \Diamond(\neg rcNotReading \land rcReading)) \]
\[ \square((rcReading \land rCs2) \rightarrow \Diamond(\neg rcReading \land rcCardOK)) \]
\[ \square((rcReading \land rCs2) \rightarrow \Diamond(\neg rcReading \land rcCardNotOK)) \]
\[ \square((rcCardOK \land rCs1) \rightarrow \Diamond(\neg rcCardOK \land rcNotReading)) \]
\[ \square((rcCardNotOK \land rCs1) \rightarrow \Diamond(\neg rcCardNotOK \land rcNotReading)) \]

% in ejectCard()

\[ \square((ecNotEjecting \land eCs2) \rightarrow \Diamond(\neg ecNotEjecting \land ecWillingToEject)) \]
\[ \square(ecWillingToEject \rightarrow \Diamond(\neg ecWillingToEject \land ecEjected)) \]
\[ \square((ecEjected \land eCs1) \rightarrow \Diamond(\neg ecEjected \land ecNotEjecting)) \]

% in cancel()

\[ \square((caNotCancelling \land cAs1) \rightarrow \Diamond(\neg caNotCancelling \land caWillingToCancel)) \]
\[ \square((caWillingToCancel \land cAs2) \rightarrow \Diamond(\neg caWillingToCancel \land caCancelled)) \]
\[ \square((caCancelled \land cAs1) \rightarrow \Diamond(\neg caCancelled \land caNotCancelling)) \]

% SUBPROCESS PRESCRIPTIONS

% by manager ATM

\[ \square(\text{atmWaiting} \rightarrow (cPs1 \land gMs1 \land eCs1 \land rCs1 \land cAs1)) \]
\[ \square(\text{atmReadingCard} \rightarrow (cPs1 \land gMs1 \land eCs1 \land rCs1 \land cAs1)) \]
\[ \square(\text{atmChekingPIN} \rightarrow (cPs1 \land gMs1 \land eCs1 \land rCs1 \land cAs1)) \]
\[ \square(\text{atmPINChecked} \rightarrow (cPs1 \land gMs1 \land eCs1 \land rCs1 \land cAs1)) \]
\[ \square(\text{atmGettingMoney} \rightarrow (cPs1 \land gMs2 \land eCs1 \land rCs1 \land cAs1)) \]
\[ \square(\text{atmPayingOff} \rightarrow (cPs1 \land gMs1 \land eCs1 \land rCs1 \land cAs1)) \]
\[ \square(\text{atmCardEjected} \rightarrow (cPs1 \land gMs1 \land eCs2 \land rCs1 \land cAs1)) \]
\[ \square(\text{atmNotOK} \rightarrow (cPs1 \land gMs1 \land eCs1 \land rCs1 \land cAs1)) \]
\[ \square(\text{atmCancelling} \rightarrow (cPs1 \land gMs1 \land eCs1 \land rCs1 \land cAs1)) \]
\[ \square(\text{atmCancelled} \rightarrow (cPs1 \land gMs1 \land eCs1 \land rCs1 \land cAs1)) \]

% by manager BankComputer

\[ \square(\text{bcWaiting} \rightarrow (cPs4 \land gMs4 \land pTs1 \land vAs1)) \]
\[ \square(\text{bcVerifying} \rightarrow (cPs4 \land gMs4 \land pTs1 \land vAs2)) \]
\[ \square(\text{bcAccountVerifiedNotOK} \rightarrow (cPs3 \land gMs4 \land pTs1 \land vAs1)) \]
\[ \square(\text{bcAccountVerifiedOK} \rightarrow (cPs3 \land gMs4 \land pTs1 \land vAs1)) \]
\[ \square(\text{bcWaitingForTransactionRequest} \rightarrow (cPs4 \land gMs4 \land pTs1 \land vAs1)) \]
\[ \square(\text{bcProcessing} \rightarrow (cPs4 \land gMs4 \land pTs2 \land vAs1)) \]
\[ \square(\text{bcTransactionProcessed} \rightarrow (cPs4 \land gMs3 \land pTs1 \land vAs1)) \]
% STATE CHANGES IN MANAGER PROCESSES

% in ATM

\[ (\neg \text{atmWaiting} \land \text{atmReadingCard} \land \neg \text{rCs1} \land \neg \text{trC1}) \rightarrow \]
\[ (\neg \text{atmWaiting} \land \neg \text{atmCheckingPIN} \land \neg \text{cPs1} \land \neg \text{rCs2} \land \neg \text{tcp1} \land \neg \text{trC2} \land \neg \text{trC3}) \]
\[ (\neg \text{atmCheckingPIN} \land \text{tcp2}) \rightarrow \]
\[ (\neg \text{atmCheckingPIN} \land \neg \text{atmPINChecked} \land \neg \text{cPs2} \land \neg \text{tcp2} \land \neg \text{tcp3}) \]
\[ (\neg \text{atmPINChecked} \land \text{tcp1} \land \text{tgM1}) \rightarrow \]
\[ (\neg \text{atmPINChecked} \land \text{atmGettingMoney} \land \neg \text{gMs1} \land \neg \text{tgM1}) \]
\[ (\neg \text{atmGettingMoney} \land \text{tcpM2}) \rightarrow \]
\[ (\neg \text{atmGettingMoney} \land \neg \text{atmPayingOff} \land \neg \text{gMs2} \land \neg \text{tgM2}) \]
\[ (\neg \text{atmPayingOff} \land \text{tcp1} \land \text{tgM1} \land \text{teC1}) \rightarrow \]
\[ (\neg \text{atmPayingOff} \land \neg \text{atmCardEjected} \land \neg \text{eCs1} \land \neg \text{teC1}) \]
\[ (\neg \text{atmCardEjected} \land \text{tcp2}) \rightarrow \]
\[ (\neg \text{atmCardEjected} \land \neg \text{atmReadingCard} \land \neg \text{eCs2} \land \neg \text{teC2}) \]
\[ (\neg \text{atmReadingCard} \land \text{tcp3}) \rightarrow \]
\[ (\neg \text{atmReadingCard} \land \neg \text{atmCheckingPIN} \land \neg \text{rCs2} \land \neg \text{trC2} \land \neg \text{trC3}) \]
\[ (\neg \text{atmCheckingPIN} \land \text{tcpP3}) \rightarrow \]
\[ (\neg \text{atmCheckingPIN} \land \neg \text{atmNotOK} \land \neg \text{cPs2} \land \neg \text{tcp2} \land \neg \text{tcp3}) \]
\[ (\neg \text{atmCheckingPIN} \land \text{tcpA1}) \rightarrow \]
\[ (\neg \text{atmCheckingPIN} \land \neg \text{atmCancelling} \land \neg \text{rCs2} \land \neg \text{cAs1} \land \neg \text{trC2} \land \neg \text{trC3} \land \neg \text{tcpA1}) \]
\[ (\neg \text{atmCancelling} \land \text{tcpA1}) \rightarrow \]
\[ (\neg \text{atmCancelling} \land \neg \text{atmNotOK} \land \neg \text{cAs1} \land \neg \text{tcpA1}) \]
\[ (\text{atmNotOK} \land \text{tcpA1}) \rightarrow \]
\[ (\neg \text{atmNotOK} \land \text{tcA2}) \rightarrow \]
\[ (\neg \text{atmCancelling} \land \neg \text{atmCancelled} \land \neg \text{cAs2} \land \neg \text{tcpA2}) \]
\[ (\neg \text{atmCancelled} \land \text{tcpA1}) \rightarrow \]
\[ (\neg \text{atmCancelling} \land \neg \text{atmNotOK} \land \text{tcpA1}) \]

\[ (\neg \text{atmNotOK} \land \text{tcpM2}) \rightarrow \]
\[ (\neg \text{atmNotOK} \land \text{tcpP3}) \rightarrow \]
\[ (\neg \text{atmNotOK} \land \text{tcpA2}) \rightarrow \]
\[ (\neg \text{atmNotOK} \land \text{tcpA1}) \rightarrow \]
\[ (\neg \text{atmCancelling} \land \neg \text{atmNotOK} \land \text{tcpA2}) \rightarrow \]
\[ (\neg \text{atmCancelling} \land \neg \text{atmNotOK} \land \text{tcpA1}) \rightarrow \]

% in BankComputer

\[ (\neg \text{bcWaiting} \land \text{tcp5} \land \text{tvA1}) \rightarrow \]
\[ (\neg \text{bcWaiting} \land \neg \text{bcVerifying} \land \neg \text{vAs1} \land \neg \text{tvA1}) \]
\[ (\neg \text{bcVerifying} \land \text{tcpA3}) \rightarrow \]
\[ (\neg \text{bcVerifying} \land \neg \text{bcAccountVerifiedNotOK} \land \neg \text{cPs4} \land \neg \text{vAs2} \land \neg \text{tcp5} \land \neg \text{tvA2} \land \neg \text{tvA3}) \]

36
\[ ((bcAccountVerifiedNotOK \land tcP4) \rightarrow
\neg (bcAccountVerifiedNotOK \land bcWaiting \land \neg cPs3 \land \neg tcP4)) \]

\[ ((bcVerifying \land tvA2) \rightarrow
\neg (bcVerifying \land bcAccountVerifiedOK \land \neg cPs4 \land \neg vAs2 \land
\neg tcP5 \land \neg tvA2 \land \neg tvA3)) \]

\[ ((bcAccountVerifiedOK \land tcP4) \rightarrow
\neg (bcAccVerifiedOK \land bcWaitingForTransactionRequest \land \neg cPs3 \land \neg tcP4)) \]

\[ ((bcWaitingForTransactionRequest \land tgM4 \land tpT1) \rightarrow
\neg (bcWaitingForTransactionRequest \land bcProcessing \land \neg pTs1 \land \neg tpT1)) \]

\[ ((bcProcessing \land tpT2) \rightarrow
\neg (bcProcessing \land bcTransactionProcessed \land
\neg gMs4 \land \neg pTs2 \land \neg tgM4 \land \neg tpT2)) \]

\[ ((bcTransactionProcessed \land tgM3) \rightarrow
\neg (bcTransactionProcessed \land bcWaiting \land \neg gMs3 \land \neg tgM3)) \]

% INSIDE TRAPS

% belonging to checkPIN()

\[ ((cPs1 \land cpNotChecking) \rightarrow tcP1) \]

\[ ((cPs2 \land (cpConnected \lor cpVerifying \lor cpChecked)) \rightarrow tcP2) \]

\[ ((cPs3 \land (cpNotChecking \lor cpConnected \lor cpChecked)) \rightarrow tcP3) \]

\[ ((cPs4 \land cpVerifying) \rightarrow tcP4) \]

% belonging to getMoney()

\[ ((gMs1 \land gmNotGetting) \rightarrow tgM1) \]

\[ ((gMs2 \land (gmConnected \lor gmProcessed \lor gmMoneyPaid)) \rightarrow tgM2) \]

\[ ((gMs3 \land (gmNotGetting \lor gmConnected)) \rightarrow tgM3) \]

\[ ((gMs4 \land gmProcessed) \rightarrow tgM4) \]

% belonging to verifyAccount()

\[ ((vAs1 \land vaNotVerifying) \rightarrow tvA1) \]

\[ ((vAs2 \land vaAccountVerifiedOK) \rightarrow tvA2) \]

\[ ((vAs2 \land vaAccountVerifiedNotOK) \rightarrow tvA3) \]

% belonging to processTransaction()

\[ ((pTs1 \land ptNotProcessing) \rightarrow tpT1) \]

\[ ((pTs2 \land ptProcessed) \rightarrow tpT2) \]
% belonging to readCard()

\[ (rcs1 \land rcReading) \rightarrow trC1 \]
\[ (rcs2 \land rcCardOK) \rightarrow trC2 \]
\[ (rcs2 \land rcCardNotOK) \rightarrow trC3 \]

% belonging to ejectCard()

\[ (ecs1 \land ecNotEjecting) \rightarrow teC1 \]
\[ (ecs2 \land (ecWillingToEject \lor ecEjected)) \rightarrow teC2 \]

% belonging to cancel()

\[ (cas1 \land caWillingToCancel) \rightarrow tcA1 \]
\[ (cas2 \land caCancelled) \rightarrow tcA2 \]

% INITIAL CONDITIONS

init

init \rightarrow (cpNotChecking \land gmNotGetting \land vaNotVerifying \land ptNotProcessing \land rcNotReading \land ecNotEjecting \land caNotCancelling \land atmWaiting \land bcWaiting)

\%........................< Here ends the PTL program >........................


7 Model verification

This section shows that it is possible to link the output of our translation, a PLTL-based program, to a verification procedure about correctness in the initial PARADIGM specification. We show that well-known properties from the systems verification literature [MP92] 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”

□ ¬(vaAccountOK ∧ vaAccountNotOK) △

Example 7 A guarantee property:
“It is possible for the ATM to report a PIN as checked while BankComputer is still verifying it”

◊(cpChecked ∧ 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.”

□(cpVerifying → ◊ cpChecked)

“If ATM requests BankComputer to verify a PIN, it always gets an answer, be it positive or negative”

□(TcP4 → ◊(TvA2 ∨ TvA3)) △

Example 9 A response/recurrence property:
“The stage of verifying account implies to check whether the account is acceptable or not. After that step the process is reinitiated.”

□(vaNotVerifying → ◊((vaAccountOK ∨ vaAccountNotOK) ∧ ◊ vaNotVerifying)) △
Example 10 A recurrence property:
“The process of checking a PIN can be cyclically invoked”

\[ \Box (\Diamond \text{cpNotChecking} \land \Diamond \neg \text{cpNotChecking}) \]

It can be seen the verification process can be set, either at the more general level of the functionality of the system (examples 6 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, e.g. ETP, 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 succesfull 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.

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. On the other hand, K. Etessami [Ete99] shows that is possible to translate an extended version of Linear Temporal Logic (LTL) to Buchi Automata, being the latter another specification language for SPIN. As PLTL is a sublogic of LTL, it is likely that further research support SPIN as a profitable tool for verifying PARADIGM models, i.e. by first using our algorithm to translate a PARADIGM model \( M \) to a PLTL program \( P \), then by using Etessami’s work to translate \( P \) (an LTL program) to Buchi automata \( B \) and finally by using \( B \) as the input for SPIN.

We found that PLTL is a flexible language where to easily encode a wide range of distinctive features in Paradigm, e.g. those relating traps and subprocesses. It is our conjecture that encoding those 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 involved with Paradigm based specifications. They hopefully will allow us to give some other steps on improving the verification possibilities available to Knowledge and Software Engineers using PARADIGM.

8 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 for tracing process inter-
actions. Also, it can be seen as a database that can be queried for system verification. 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. Indeed, we have been testing a current implementation in
PROLOG.

A very interesting issue to be considered in further work involves rule enhancement for modelling
processes that are not always active, as it is usually the case for real systems. Translation could be
also extended for expressing some constraints that are not included in PARADIGM models but
usually affects 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 (see e.g. Fig. 2).

9 Acknowledgments

The authors wish to thank Silvia Teresita Acuña, Mabel Sosa, Pablo Fillottrani and Elsa Estevez
for their feedback about the problem addressed in this work.
References


